Readable s-expressions and sweet-expressions: Getting the infix fix and fewer parentheses in Lisp-like languages

dwheeler.com (https://www.dwheeler.com/readable/readable-s-expressions.html)

by David A. Wheeler (http://www.dwheeler.com), 2006-06-17 (revised 2008-09-04)

This page is obsolete; see http://readable.sourceforge.net (http://readable.sourceforge.net) instead.

Many people find Lisp s-expressions hard to read as a programming notation. This paper discusses various ways to extend/modify s-expressions so they can be more readable without losing their power (such as quasiquoting, macros, and easily-manipulated program fragments). The goal is a notation that can be trivially translated to and from traditional s-expression notation (both by computer and in people's heads), and isn't dependent on a particular underlying semantic. The paper identifies and discusses three approaches that seem particularly promising: indentation, name-prefixing (so func(x y) is the same as (func x y)), and infix support.

It then defines a particular way of combining these approaches, called "", that can be viewed as an essentially backward-compatible extension of s-expressions. A sweet-expression reader can accept typical cleanly-formatted s-expressions without change, but it also supports various extensions (optional syntactic sugar) that make much clearer code possible. This is purely a matter of screen presentation; underlying systems can continue to use s-expressions, unchanged. For example, here's a trivial Common Lisp program that takes advantage of sweet-expression's formatting extensions (the Scheme version is similar):

```
defun factorial (n)         ; Parameters can be indented, but need not be
  if (n <= 1)               ; Supports infix, prefix, & function <=(n 1)
      1                     ; This has no parameters, so it's an atom.
      n * factorial(n - 1)  ; Function(...) notation supported
```

Sweet-expressions add the following abilities:

1. **Indentation**. Indentation may be used instead of parentheses to start and end expressions: any indented line is a parameter of its parent, later terms on a line are parameters of the first term, lists of lists are marked with GROUP, and a function call with 0 parameters is surrounded or followed by a pair of parentheses. A "(" disables indentation until its matching ")". Blank lines at the beginning of a new expression are ignored. A term that begins at the left edge and is immediately followed by newline is immediately executed, to make interactive use pleasant.
2. **Name-prefixing**. Terms of the form 'NAME(x y...)', with no whitespace before '(', are interpreted as '(NAME x y...)';. If its content is an infix expression, it's considered one parameter.
3. **Infix**. Optionally, expressions are automatically interpreted as infix if their second parameter is an infix operator (by matching an "infix operator" pattern of symbols), the first parameter is not an infix operator, and it has at least three parameters. Otherwise, expressions are interpreted as normal "function first" prefix notation. Infix expressions must have an odd number of parameters with the even ones being binary infix operators. You must separate each infix operator with whitespace on both sides. You can chain the same infix operator, so (2 + 3 + 4) is fine; to mix infix operators, use parentheses. Thus "2 + (y * -(x)" is a valid expression, equivalent to (+ 2 (* y (- x))). Infix operators must match this pattern (and in Scheme cannot be =>):

```
    [+-\*/<>=&\|\p{Sm}]{1-4}|\:|\|\|
```

For more information, see my website at http://www.dwheeler.com/readable (http://www.dwheeler.com/readable).

This paper describes the rationale behind the older sweet-expressions version 0.1; see Sweet-expressions: Version 0.2 (https://www.dwheeler.com/readable/version02.html) for information on the changes made to sweet-expressions since this paper.

Introduction

S-expression notation is a very simple notation for programs, and programs in variants of the Lisp programming language have traditionally been written using s-expressions. In this notation, an operation and its parameters is surrounded by parentheses; the operation to be performed is identified first, and each parameter afterwards is separated by whitespace. So "2+3" is written as "(+ 2 3)". As noted in the May 2006 Wikipedia, this syntax "is extremely regular, which facilitates manipulation by computer. The reliance on [s-]expressions gives the language great flexibility. Because Lisp functions are themselves written as lists, they can be processed exactly like data: allowing easy writing of programs which manipulate other programs (metaprogramming)." In short, s-expressions are a powerful and regular way to represent programs and other data.

I've written a lot of Lisp code, so I've learned to read s-expressions fairly well. (I wrote a *lot* of Lisp code in the late 1980s on a $120,000 system.) But I am never the only one who reads my programs -- I need to make sure others can read my programs too.

Here's the problem: for most software developers, programs written solely using s-expressions are hard to read, and they will *only* voluntarily use programming languages that allow, at least optionally, a more common notation. This is particularly true for the usual infix operations (+, <, and so on). People who use Lisp-based languages all the time eventually learn, but not everyone *wants* to use them all the time, and even developers who are comfortable with programs in s-expression notation need to share their work with others. Wikipedia notes that “the heavy use of parentheses in S-expressions has been criticized -- some joke acronyms for Lisp are ‘Lots of Irritating Superfluous Parentheses’, ‘Let’s Insert Some Parentheses’, or ‘Long Irritating Series of Parentheses’ “.

Yes, I know the arguments. “S-expressions are powerful and regular” ! Of course they are. They are a wonderful intermediate representation for lots of things, in fact. But they are a *terrible* user interface, especially if you are trying to share your results with others. People today -- even most programmers -- want systems that are “easy to use”, and one of the best ways to make something easy to use is to make it *familiar*. Most software developers have been trained for *many years* to use traditional infix mathematical notation, and S-expression notation fails to use it. Programming is often not a solo effort; development today is practically always a group effort, and readability for a large diverse group matters.

All the loud statements about the power of S-expressions cannot compete with time looking at real programs. Most software developers laugh at languages that are “obviously weak” (to them) because their default parser cannot handle <, *, and - in conventional ways. Below is a trivial Common Lisp program to compute a factorial - now ask, “is this really the most readable format possible for other readers?”

```
(defun factorial (n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))
```

Maybe you think so now, but I use this trivial program as an example to show some alternatives that I think many people would prefer. For example, here's the same program, but exploiting the abilities of ; a reader for sweet-expressions could read both the previous expression and this one:

```
defun factorial (n)
  if (n <= 1)
      1
      n * factorial(n - 1)
```

The 1993 paper "The Evolution of Lisp" by Guy L. Steele, Jr. and Richard P. Gabriel (http://citeseer.ist.psu.edu/steele93evolution.html) section 3.5.1 discusses "Algol-style Syntax", give a history, and slyly makes fun of efforts to try to use anything other than S-expressions: "Algol-style syntax makes programs look less like the data structures used to represent them. In a culture where the ability to manipulate representations of programs is a central paradigm, a notation that distances the appearance of a program from the appearance of its representation as data is not likely to be warmly received… it is always easy for the novice to experiment with alternative notations. Therefore we expect future generations of Lisp programmers to continue to reinvent Algol-style for Lisp, over and over and over again, and we are equally confident that they will continue, after an initial period of infatuation, to reject it. (Perhaps this process should be regarded as a rite of passage for Lisp hackers.)" Paul Graham posts a longer version of this quote (http://www.paulgraham.com/syntaxquestion.html).

Perhaps. But it's worth noting that thousands of languages have been invented over the years, but almost none decide to use S-expressions as their *surface* expressions. Smalltalk and Python took many ideas from Lisp (see Norvig's "Python for Lisp programmers" (http://www.norvig.com/python-lisp.html) for more)… but not its surface syntax. Languages like ML and Haskell have strong academic communities… and do not use S-expressions for their surface syntax either. Logo was devised to have Lisp's power, but intentionally chose to not use its syntax. Dylan actually *switched* from S-expressions to a more traditional syntax, when they wanted "normal" people to use it. Yacas (http://yacas.sourceforge.net/codingmanual.html) is intentionally Lisp-like underneath, but completely abandoned Lisp's surface syntax for normal user interaction (as have essentially all computer algebra systems, even though many have a Lisp underneath).

The fact that so many people are compelled to find a "fix" to Lisp syntax indicates to me that there is a *problem*… the fact that so many efforts fail suggests that it is a *hard* problem. Norvig's Lisp retrospective (http://www.norvig.com/Lisp-retro.html) found that even in 2002, when comparing Lisp to Python and Java, Lisp was *far* faster and more extensible, and had many powerful properties… but it was gaining few users (it was mostly stagnant). I argue that one key reason is the syntax; Lisp's syntax looks genuinely hostile to most "normal programmers". Instead of Lisp-like languages gaining converts who adjust to prefix syntax, many people are ignoring or abandoning Lisp-like languages to use languages designed to be *readable* by humans. John Foderaro correctly said, "Lisp is a programmable programming language", but people expect standard notation to be *already available*. Obviously it's easy to create a some kinds of front-end syntax for Lisp-like systems, but they generally don't catch on -- in part because they usually don't provide the full power of the system underneath (e.g., you often can't do backquoting with comma-lifting, or insert new parameters in a control structure, or handle macros well). I prefer to use the best tool for the job; when it's not Lisp, don't use Lisp. But some jobs are naturals for Lisp-based langauges, yet their poor readability seriously interferes with that. I think there are ways to retain the power of Lisp-based languages, while improving their readability and retaining their flexibility.

Oh, and let's debunk one claim here. Steele and Gabriel claim one problem with Algol-like syntax is that there are "not enough symbols". Yet even if that were true, by combining characters lots of symbols can be created without resorting to fancy characters. Indeed, <= is a common combined character for "less than or equal to"; nobody has trouble understanding that.

So below, I discuss a lot of past and current work to improve the syntax of Lisp-like languages. I then focus on a few especially promising areas that can make Lisp more readable while still accepting normal s-expression notation *and* having only minimal changes to it. These include:

1. for meaning (like Python and Haskell); there is already work on this, particularly with I-Expressions.
2. , so f(x) and (f x) mean the same thing. Norvig earlier experimented with this.
3. - how in the world can we rationally add infix support? There are lots of options, which I explore.

After looking at many of the possibilities, I then present "", which I believe is an approach that resolves these problems in a reasonable way. I then end with . In an appendix I document BitC a little further, since it started the whole thing for me, and in a separate paper on alternatives for s-expressions (https://www.dwheeler.com/readable/alternative-s-expressions.html) I document some of the alternatives I tried before I came up with s-expressions.

Goals

My goals were originally for the BitC project, but now I think there might be ways to work more widely for any language that uses Lisp-like notations. Here were my goals for a programming notation for Lisp-like systems:

1. Readable. It should be more “readable” to the uninitiated, in particular, it should look like more traditional notation. For example, ideal format would support infix notation for operations that normally use infix (+, -, <=, etc.), support having function names before parentheses, and not *require* as many parentheses.
2. Mappable. There needs to be an obvious mapping to and from current s-expression notation, which must work for both data and code. The key advantage of Lisp-like languages is that you can easily manipulate programs as data and vice-versa; that *must* remain for any modified syntax. Otherwise, there’s no point; there are lots of other very good programming languages that support infix and other nice notations.
3. General/Standardizable. It should be very general and standardizable across all systems that accept s-expressions (Lisp's original syntax); nobody wants to relearn syntax everywhere. It should be useful in Common Lisp, Scheme, Emacs Lisp, BitC, ACL2, DSSSL, AutoLISP (built into AutoCAD), ISLISP (standardized by ISO) BRL (http://brl.sourceforge.net/), and so on. A thread about guile noted this very need (http://sourceware.org/ml/guile/2000-07/msg00160.html).
4. Easily implemented (relatively speaking). It must not require tens of thousands of lines of code to do. However, if it takes a little extra code to produce nice results, that is fine; better to do things well once. It need not be easily implementable via a few tweaks of an existing reader, though that’d be nice. Yes, rewriting a reader is a pain, but those only have to be written once per implementation. Note that even among implementation of a particular language there is often much variance, so the format needs to be simple enough to support many implementations.
5. Quote well. In particular, for both forward quote (’) and backquote/quasiquote (‘), it must be easy to find the end of the quote, and for backquote/quasiquote, it would be very desirable to support initial comma (,) and friends to locally reverse the quoting (the whole point of quasiquoting).

6. Backward-compatible. Ideally, it should be able to read regular s-expressions (at least normally-seen formats of them) as well as the extensions. I’m willing to give a little on this one where necessary. *Note: Version 0.2 of sweet-expressions is not perfectly backward-compatible, but most typical s-expressions as people actually use them are also valid sweet-expressions*.
7. Work with macros. There will probably be some tweaks for indenting and infix notation (especially since infix reorders things!), but macro processing should continue to work in most cases.

My notion is that the underlying s-expression system would not change… instead, the system would support a reader that takes an extended notation and converts it into s-expressions. A printer could then redisplay s-expressions later in the traditional notation, or same kind of notation used to input it. It would also be nice if the notation was not confusing or led to likely errors.

I’ve written this document to put at least a few ideas down in writing. In the long term, I suspect what needs to happen is for there to be some sort of “neutral” forum where ideas can be discussed, and code shared. For any syntax to be widely used, there must be trustworthy, widely-usable implementations. I think at least one FLOSS implementation with a generous license that permits use by proprietary programs and Free-libre / open source software (FLOSS) (http://www.dwheeler.com/oss_fs_why.html) programs. Note that the LGPL doesn’t work as intended with most Lisp implementations; Franz has created the Lisp LGPL (LLGPL) (http://opensource.franz.com/preamble.html) which is a specific clarification of the LGPL for use with Lisp. Lisp code is typically licensed under the LLGPL instead of the LGPL (http://common-lisp.net/faq.shtml), since the LLGPL clarifies some otherwise sticky issues and ambiguities in the LGPL. Note that any FLOSS software should be GPL-compatible (http://www.dwheeler.com/essays/gpl-compatible.html), since there is so much GPL'ed code. The implementations would need to be widely portable and modular, so that they can be widely depended on.

Scheme and Common Lisp aren't really compatible at all. Translators like scm2cl (Scheme to Common Lisp translator) (http://www.ccs.neu.edu/home/dorai/scmxlate/scm2cl.html) could help initially, certainly to get started (suggesting that it'd be best to start with Scheme, and then transition at least an initial version to Common Lisp). But in the end, specialized implementations well-tuned to different environment will be necessary for an improved syntax to really work.

There are lots of Lisp code repositories and other Lisp-related sites, including Common-Lisp.net (http://common-lisp.net/), Lisp.org / Association of Lisp users (ALU) (http://lisp.org/), CLiki (Common Lisp Wiki) (http://www.cliki.net/index), and schemers.org (http://schemers.org).

Past Work

Here are some past efforts to try to make s-expressions more readable.

Special syntax for various constructs

Most readability efforts focus on creating special syntax for every language constructs; these often end up unused (because they cannot keep modifying the grammar to match the underlying system), or end up creating a completely new language less suitable for self-analysis of program fragments.

“The Evolution of Lisp” (http://citeseer.ist.psu.edu/steele93evolution.html) lists many efforts to create "Algol-like notations for Lisp", which generally included infix notations and attempts to be more “readable”. Originally Lisp was supposed to be written in M-expressions, which were more traditional in format. Function calls were written as F[x;y...] instead of (F x y...), for example. One problem is that they kept adding new syntax, and never found a good “final” M-expression format, so it was never implemented... and it just receded into the never-finished future. Many other notations were developed, including those of LISP 2. These generally had if ... then ... else and other more traditional naming conventions.

For example, RLisp (http://portal.acm.org/ft_gateway.cfm?id=1089393&type=pdf&coll=portal&dl=ACM&CFID=15151515&CFTOKEN=6184618) (used by Reduce, among others) is a Lisp with an infix notation. A yacc grammar for RLisp by A. C. Norman (2002) (http://gauguin.trin.cam.ac.uk/csl/util/r2l.y.old) is available. Norman notes that the grammar is “ambiguous or delicate in several areas”:

> 1. It has the standard “dangling else” problem.
> 2. If R is a word tagged as RLIS, then R takes as its operands a whole bunch of things linked by commas. At present I have this grammar ambiguous on R1 a, b, c, R2 d, e, f; where R2 could (as far as the grammar is concerned) be being given one, two or three arguments. This problem arises if the operands of R may themselves end in an R. This is harded to avoid than I at first thought - one might well want conditionals in the are list of an R, but then R1 a, IF x THEN R2 b, c; comes and bites. I guess this is a “dangling comma” problem. The above two problems are resolved by the parser genarator favouring shift over reduce in the ambiguous cases.
> 3. “IN”, “ON” are both keywords, as used in for each x in y do ... and words with the RLISTAT property. This is sordid! Similarly “END” has a dual use.

This is coped with by making special provision in the grammar for these cases.

One trouble of many of these notations is that it becomes difficult to see where the end of an expression is (e.g., there might be no way to indicate the “end” of the if statement); this can creates ambiguities and makes it harder to easily match the infix notation and s-expression if you need to. And, if you want to describe arbitrary Lisp s-expressions, not having an end-marker means that you may not able to access some of the capabilities of the underlying s-expressions (though that may not be an issue for all uses).

Using a hierarchy of Domain Specific Languages in complex software systems design (http://arxiv.org/abs/cs.PL/0409016) by V. S. Lugovsky discusses using Lisp to create domain-specific languages, including syntactic transformations.

The ACL2 language is Common Lisp-based, but it has a separate front-end for a more traditional interface including an infix processor; it is named IACL2 (http://www.cs.utexas.edu/users/moore/infix/printer/syntax.html). This is not often used, for several reasons:

1. ACL2 is only defined in s-expression form, so you cannot read any of the documentation or examples without knowing s-expression form anyway.
2. IACL2 is not as portable as ACL2, nor as supported.
3. IACL2 does not support all the capabilities as ACL2 -- and who wants to use a tool that is known to not work when you most need it?

IACL2 has a “dangling else”, so it is not always trivial to see where something ends.

Logo (http://en.wikipedia.org/wiki/Logo_programming_language) is basically Lisp with an infix and more readable syntax. Instead of “(”…”)”, Logo uses “[”…”]”. Normally, all commands begin with the name of the function, just like Lisp, and

Logo even has text names for math functions: sum, product, difference, and quotient. More interestingly, infix is also available… by using symbols, Logo automatically uses the infix forms instead. Logo knows the number of parameters for each function, so once the number of parameters is provided you can just provide another function call on the same line without any marking of the end of the call (see the “rt” call below). This is not as flexible as s-expressions, where you can always add another parameter. Here’s a sample Logo program:

```
to spiral :size
   if  :size > 30 [stop] ; a condition stop
   fd :size rt 15        ; many lines of action
   spiral :size *1.02    ; the tailend recursive call
end
```

Dylan (http://dylanpro.com/DylanExchange.html) is another Lisp with more conventional notation, including an infix format. Here’s a Lisp-to-Dylan translator, exploiting the Common Lisp pretty-printer (http://www.norvig.com/ltd/doc/ltd.html). D-Expressions: Lisp Power, Dylan Style (http://people.csail.mit.edu/jrb/Projects/dexprs.pdf) even shows it is possible to combine infix forms with Lisp’s abilities to manipulate programs. Dylan is a little wordy, in part because of namespace problems (types are in the same namespace, which is often not what you want), but it’s easy to read. It ends blocks with “end blockname”, e.g., “if (a) b else c end if”; a little long but clear. Here’s a simple example from the page Procedural Dylan (http://www.webcom.com/haahr/procedural-dylan/1-distance.html):

```
define method distance (x1 :: <real>, y1 :: <real>, x2 :: <real>, y2 :: <real>)
  => distance :: <real>;
   sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1))
end method distance;
```

There is also CGOL (an Algol-like language that compiles into Common Lisp) (http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/code/syntax/cgol/0.html); this was developed originally by Vaughn Pratt, and written by a UC Berkeley graduate student, Tom Phelps. This program is a Common Lisp implementation of CGOL that is translated into Lisp before execution. I do not know what its license is. You can look more generally at the CMU archive (http://www-cgi.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/code/syntax/0.html), though beware of licensing problems.

Clike (http://www.rojoma.com/~robertm/clike/) (see also) is a compiler which converts code written in a simple C-like language to Scheme. Pico (http://pico.vub.ac.be) is an educational Scheme with C-ish syntax.

TwinLisp (http://twinlisp.nongnu.org/docs/TwinLisp%20for%20lisp%20users.html) is a "new way of programming in Common Lisp"; it creates a new reader with a more conventional syntax, which it then translates to Common Lisp and executes. It predefines precedence - which unfortunately means that it is tied to a particular interpretation of all operations. It also defines special control structures; thus, like most past efforts to create "readable Lisp", it ends up sacrificing generality of the resulting format (the reader is rigged to a specific set of operators with special syntax - so creating new operators is not simply a matter of defining the meaning of the operator).

Lisp resources (http://www.lisp.org/alu/res-lisp-tools) gives pointers for where to go for more information. For extensions specifically focused on indentation, see the text below on indentation; for extensions specifically focused on infix notation, see the section on infix.

Skill

Skill (http://portal.acm.org/ft_gateway.cfm?id=174185&type=pdf&coll=portal&dl=ACM&CFID=72577012&CFTOKEN=50239143) from Cadence is a proprietary Lisp-based extension language. By examining that paper, this text (http://www.thecwlzone.com/Professional/Programming/skill.html), and this document (http://wrcad.com/manual/xicmanual/node451.html), a hint of its syntax can be inferred. Skill supports name-prefixing, where FUNC(x y) is treated as meaning (FUNC x y). Skill also supports infix processing, automatically translating infix operators into an internal prefix format. In Skill, it appears that all of the prefix functions have alphabetic names, e.g., "(plus 3 4)" adds 3 to 4. This means that infix operators can be unambiguously detected and used without surrounding whitespace; thus "3+4" is automatically translated to "(plus 3 4)". Skill does not consider indentation significant.

I point out Skill specially because it does shows that it's possible to have a very general syntax that is easier to use.

Ioke

Ioke (http://kenai.com/projects/ioke/) is a language developed by Ola Bini, a core JRuby developer. An interview with Ola Bini about Ioke (http://www.infoq.com/news/2008/11/ioke) mentions homoiconicity as important.

Arc and its influence: Syntax as Abbreviation

Paul Graham began devising a Lisp variant named Arc, and promulgated the term Syntax as abbreviation (http://www.paulgraham.com/arcll1.html) for his approach. Here's what he said:

- Common Lisp and Scheme only directly support s-expressions; "disadvantage: long-winded".
- Dylan and Python have s-expressions hidden underneath, "disadvantage: macros unnatural".

Instead, he wants Arc to have a syntax, because it makes programs shorter (a big win), but have additional notation as simply an abbreviation for the longer (still valid) syntax. He also would like to show structure by indentation instead of parentheses (which would become optional where no ambiguity results). His ideas sparked many others to re-investigate s-expression syntax, generally thinking in terms of "syntax as abbreviation".

Arc has inspired many people to re-evaluate Lisp syntax, and see if there are ways it can be made more readable. Some comments on Arc (http://www.archub.org/arcsug.txt) made some interesting points:

- Peter Norvig said, "For the Scheme I wrote for Junglee's next-generation wrapper language, I allowed three abbreviations: (1) If a non-alphanumeric symbol appeared in the first or second element of a list to be evaled, then the list is in infix form. So (x = 3 + 4) is (= x (+ 3 4)), and (- x * y) is (* (- x) y). And (2), if a symbol is delimited by a "(", then it moves inside the list. So f(a b) is (f a b), while f (a b) is two s-exps. And (3), commas are removed from the list after infix parsing is done, but serve as barriers to infix beforehand, so f(a, b) is (f a b), while in f(-a, b + c), each of -a and b + c gets infix-parsed separatly, and then they get put together as (f (- a) (+ b c)). This seemed to satisfy the infix advocates (and annoy some of the Scheme purists). You might consider something like this." Here are some of my thoughts about this:

    - Rule 1 (the check on the first and second elements of the list) is a little hackish, and by having a check on the first element too, you can't use usual Lisp notation at all for infix operators.
    - Rule 3 (removing commas) is simple. Amazingly enough, this doesn't interfere with quasiquoting *as long as* commas are *always* required as function parameter separators; since you cannot have a null-length parameter, any parameter beginning with a comma has a comma for lifting! But that doesn't mean that the resulting code is easy to read; all those extra commas can make it hard to find the comma that is doing the lifting, and unless prefix operators are allowed, they aren't needed for parameter separation… they just create more syntactic noise. So I haven't gone further with this, though others could.
    - Rule 2 is a neat idea; there's the minor risk of bad spacing causing problems, but it is a trivial way to get more traditional functional notation without harming the syntax (functions cannot have "(" in their name anyway). Indeed, I think there's a good idea here, as I discuss below in "Name-prefixing".
- Sudhir Shenoy's "ideas from Perl" said: "Please don't use parentheses for s-expressions ('[' and ']' or even '{' and '}' would be preferable. The biggest complaint I have about parentheses is that it makes infix mathematics (which you are planning to introduce in Arc) really hard to read. e.g. (def interest (x y) (expt (-r * (t2 - t1) / (days-in-year)))) reads really badly when compared to [def interest (x y) [expt (-r * (t2 - t1) / [days-in-year])]]. The overloading of meanings of parentheses (grouping + s-expression) which isn't a problem in Lisp may cause a loss of readability in Arc."

Kragen Sitaker's email "two-dimensional syntax for Lisp" (http://lists.canonical.org/pipermail/kragen-tol/2002-January/000666.html) summarizes some ideas from Paul Graham and Arc, but the author seems to add his own points too. He says:

- ″Paul Graham notes that you can infer some parens in Common Lisp from indentation and newlines. In particular, if a line contains more than one s-expression, it must be a prefix of a list of those s-expressions. If a sequence of lines is indented from the previous text line, it must be a continuation of that line's s-expression… All of this leads to needing extra parens around invocations of parameterless methods. Oh well.” He explicitly assumes that adding parentheses will *not* disable indentation processing.
- ″In general, infix-to-prefix transformation should be relatively simple, reversible, and allow mixing of infix and prefix expressions; infix operators can't be infix if they're the first element of a list anyway. However, infix expressions could be valid prefix expressions; how about (reduce + mylist), for example? OCaml solves this problem by adopting the infix interpretation where there is ambiguity, requiring parens around infix operators used as values. This probably isn't the best solution for a Lisp, but you could probably require (function +) instead of (+) and get away with it. I'd like to be able to use at least (+ - * ** / mod) infix, with their usual precedence; and a left-associative : for cons would shorten a heck of a lot of Lisp programs.”
- “On multiline string syntax: it's ridiculous to include the leading spaces on successive lines. Spaces to get subsequent lines of the string in to where the first line of the string started should not be included in the string; their absence should simply be a syntax error.”
- “These syntax rules still don't help cond much, either the Arc variant of cond or the traditional one.” He suggests borrowing McCarthy's syntax from the 1960s: “condition => consequent”. I (Wheeler) think that using “=>” this way would be a bad idea, at least in Scheme; Scheme already uses “=>” for a completely different meaning inside “cond” constructs. Another operator could work, though finding a clear one may be more difficult. Often “->” means “implies”, and having to similar-looking operators with different semantics is a bad idea anyway. Actually, I think that when you add

indentation (see below), constructs like “cond” actually look pretty reasonable, so this is may be best solved a different way.

A conversation about Arc and related ideas is posted on reddit. (http://programming.reddit.com/info/bstl/comments/)

BitC version 0.9 uses this “syntax as abbrevation” approach. You can indicate that “Value” has type “Type” using the quasi-function “the”, e.g., as (the Type Value), but the preferred form is “value : type”. It does not go far right now, though; + is still prefix (not infix).

Infpre (http://folk.uio.no/jornv/infpre/infpre.html) is a Common Lisp infix utility (LGPL license).

Quack (http://cvs.codeyard.net/Quack/) is yet another Lisp derivative. This adds two syntactic items. First, "postfix colon syntax" - colon with indentation as a shortcut for parentheses. Second, "infix colon syntax" - a colon without spaces around it are a shortcut for function/macro call, right-associative, so cdr:cdr:obj is (cdr (cdr obj))). He doesn't know if they're workable, and I don't find the examples convincing. Indentation has promise, but the colon ends up in a bad place and it's completely unnecessary. And while an infix operator like the infix colon is useful, making that the *only* infix operator is rather weak; where's 3 + 4?

Lisp sans parentheses hell (http://web.mac.com/srikumarks/Site/programming/Entries/2006/7/29_Lisp_sans_(((paren-theses_((hell))))).html) discusses some similar ideas:

```
1. We use () to indicate grouping. ((((a)))) will mean the same thing as (a).
2. We use right associative infix operator notation -
        a * b + c  translates to (* a (+ b c)) in lisp.
3. We use indentation (tab-size = 4) to convey grouping -
        say hello         becomes  (say hello (to lisp))
                to lisp
4. We use square brackets notation to encode lists - for instance -
           [a,b,c] means     (a b c)
    The items in a list can also be separated by line breaks. So
     [  one
        two            means     (one two three)
        three ]
    Since indentation is significant,
    [
        one
            two                    means ((one two) three)
        three
    ]

There are a few special operators that help reduce the need for
parentheses and aid readability -
1.      a : b  translates to  (a . b) and due to right associativity,
        a : b : c translates to (a b . c)
2.     a :: b translates to (a b)
3.     a := b translates to (define a b)

Here are the cases where you won't need parentheses -
1.     Your expression starts on its own line. In this case,
       the first term is the head term and the rest of the terms
       on the same line are tail terms.
2.    Your expression is the second argument to an infix operator.
      For example -
        mid :: quotient (low + high) 2
            is equivalent to
        mid :: (quotient (low + high) 2)

Exceptions -
1.    Use func() notation to indicate that you mean (func) (a function
call) and not just the value of func.

Limitations of current implementation -
1.    Back-quoting not supported.
2.    The use of comma separator outside of list expressions is not
defined and handled well enough.

Enjoy! ... and, for the record, I'll stick to the parentheses,
thank you :)
```

This aids readability, but note the inability to handle all of Lisp's capabilities (like back-quoting), and that the creator won't use it (clearly a sign that it's not good enough).

So now, let’s turn to three areas that have great promise, and then see if we can combine them together.

Lisp programs are normally presentated using indentation; the Common Lisp standard even includes a pretty printer! Experienced Lisp programmers eventually stop seeing the parentheses, and see the structure of a program instead as suggested by indentation. So why not use indentation to identify structure, since people do anyway, and eliminate many unnecessary parentheses?

Other languages, including Python, Haskell, Occam, Icon, use indentation to indicate structure, so this is a proven idea. Other recently-developed languages like Cobra (http://cobralang.com/docs/python/) (a variant of Python with strong compile-time typechecking) have decided to use indentation too, so clearly indentation-sensitive languages are considered useful by many. In praise of mandatory indentation (http://okasaki.blogspot.com/2008/02/in-praise-of-mandatory-indentation-for.html) notes that it can be *helpful* to have mandatory indentation.

Past work on indenting to represent s-expressions

Paul Graham (developer of Arc) is known to be an advocate of indentation for this purpose. As I noted above, Kragen Sitaker’s notes on Graham and Arc (http://lists.canonical.org/pipermail/kragen-tol/2002-January/000666.html) discusses how indentation can really help (in this notation, functions with no parameters need to be surrounded by parentheses, to distinguish them from atoms - “oh well”). Graham's RTML (http://en.wikipedia.org/wiki/RTML) is implemented using Lisp, but uses indentation instead of parentheses to define structure. RTML is a proprietary programming language used by Yahoo!'s Yahoo! Store and Yahoo! Site hosting products, though Yahoo are transitioning away from it. Paul Graham's comments about the RTML language design (http://lib.store.yahoo.net/lib/paulgraham/bbnexcerpts.txt) and this introduction to RTML by Yahoo (http://lib.store.yahoo.net/lib/ytimes/rtmlintro.pdf).

Darius Bacon's "indent" file (http://www.accesscom.com/~darius/), includes his own implementation of a Python/Haskell-like syntax for Scheme using indentation in place of parentheses, and in that file he also includes Paul D. Fernhout's implementation of an indentation approach. Bacon's syntax for indenting uses colons in a way that is limiting (it interferes with other uses of the colon in various Lisp-like languages). I have not had a chance to examine Paul D. Fernhout's yet. (It also includes an I-expression implementation.) All of the files are released under the MIT/X license. (Darius Bacon also created mlish, an infix syntax front end listed earlier). Lispin (http://www.lispin.org/) discusses a way to get S-expressions with indentation.

I-expressions

I-expressions (http://srfi.schemers.org/srfi-49/srfi-49.html) are an alternative method for presenting s-expressions (either program or data), using indentation. They are defined in SRFI ("surfie") 49; this has final status, making I-expressions a quasi-official part of Scheme. I-expressions have no special cases for semantic constructs of the language. SRFI 49 includes a sample implementation with an MIT-style license (based on the Sugar (http://redhog.org/Projects/Programming/Current/Sugar/) project).

Here’s an example, quoted from the Scheme Requests for Implementation (SFRI) number 49 (this example uses Scheme’s “define” function, not Common Lisp’s “defun” function):

```
define
  fac x
  if
    = x 0
    1
    * x
      fac
        - x 1
```

I-expressions can include traditional s-expression representation too; here's an example (using Scheme):

```
 define (fac x)
   if (= x 0) 1
     * x
       fac (- x 1)
```

As you can probably guess, both of these are equivalent to this traditional s-expression representation:

```
(define (fac x)
  (if (= x 0)
    1
    (* x (fac (- x 1)))))
```

The keyword “group” is used to begin a list of lists. (One minor drawback: it's more difficult to use a function named "group", so it's best to simply not create such a function.) You can drop back several indentation levels without dropping them all:

```
let
  group
    foo
      + 1 2
     bar
      + 3 4
   + foo bar
```

The SFRI permits both tabs and space characters. I-expressions can also be quoted (including being quasi-quoted); see the proposal for information. Note this means that ' followed by whitespace is now the beginning of a quote, because the initial whitespace is significant, and newlines become important.

In any indentation system for s-expressions, you need to be able to express s-expressions such as (f a), (g (h 1)), and (j (k)). In the first case, the first parameter is simple symbol, in the second case, the first parameter is a call with at least one parameter, and in the third case, the call to k has no parameters at all. I-expressions represent 0-parameter function calls by having the parameter surrounded with (…) (or having () as a name-ender). Let's see what the indentation rules by themselves look like:

```
; Here is (f a):
  f
    a
; or
  f a

; Here is (g (h 1)):
  g
    h
      1
; or
  g
    h 1
; or
  g (h 1)

; Here is (j (k)), note the treatment of 0-parameter calls:
  j
    (k)
; or
  j (k)
```

The idea is simple, but it can be hard to reason about how it works -- so here is one way to think about it (if you're using this to write a program). On any line, the first term is the function to be called; the rest of the items on the line, and each indented line below, are its parameters. If a line has no other parameters (on the rest of the line or indented), then it's an atom; otherwise, it's treated as one expression and parentheses surround that line and its indents (if any). The “group” term creates an extra surrounding (...) in the resulting s-expression, so you can create lists of lists.

So how do you present computing a function and *then* calling it? Personally, I’d just switch to traditional s-expression notation in this case. But the I-expression representation actually isn’t bad; just use a “group” followed by how to compute the function to be called:

```
; ((getfunction x) a b) can be represented as:
   group
     getfunction
       x
     a
     b
; or:
   group (getfunction x)
     a
     b
```

The SFRI supplies Guile code (not fully portable Scheme code) to implement I-expressions (e.g., it uses define-public and not define). However, it should be easy to port.

In the I-expression sample implementation a “(” disables I-expression processing until its matching “)”, and this is an intended part of the definition of I-expressions. This has all sorts of wonderful side-effects:

1. I-expressions parsing becomes very safe to use with existing code - pre-existing oddly-indented code will almost certainly start each expression with an opening parenthesis, disabling indentation processing.
2. It supports dealing with text that is very close to running off the right-hand side; just use parentheses to disable indentation processing. Python does the same thing.

Disabling indentation this way doesn’t interfere with the use of parentheses to invoke function calls with 0 parameters - such calls don’t take that much room on a line!

There are a few special cases where the reader with I-expressions enabled parsed a file differently. When two top-level s-expressions follow each other, either (1) on the same line or (2) on consecutive lines with the second expression indented, the lines will be combined by an I-expression reader. The first seems unlikely to me; the second is a little unfortunate. What's worse, though, is that in a read-eval-print loop, users have to enter return twice to see their results, and it's easy to end up trying to evalute an empty list. We'll discuss those below in possible extensions to I-expressions.

The mailing list discussion of I-expressions includes a posting of an I-expression pretty-printer (http://srfi.schemers.org/srfi-49/mail-archive/msg00008.html); the author says "I just couldn't get to sleep, so I wrote one". Be warned: I believe this pretty-printer has a bug, because it does not handle (f (g)) correctly. The first pretty-printer I wrote for I-expressions in Common Lisp had the same error, too, so this is a common mistake when doing this task - be warned. The expression (f (g)) should print like this:

```
f
 (g)
```

I believe that users who use indenting expressions should *only* use space characters, and never tabs, to indent. The R5RS Scheme specification doesn't even officially permit tab characters (though implementations generally do). The real problem is that tabs produce a varying number of spaces on real systems; experience with Python suggests that using tabs can cause a lot of problems. Most of today’s text editors can be configured to turn tabs into spaces automatically.

Possible extensions to I-expressions

I-expressions and similar indentation approaches are a very nice approach. But there are several ways it could be extended or modified, now that I've had a chance to experiment with them.

Immediate execution if begin on left-hand-edge

If you use indented I-expressions interactively (in a read-eval-print loop), you have to enter an extra a blank line (an extra "enter") before the expression is evaluated. That's because the system needs to know when an expression has ended; the system believes you might indent the next line and add more expressions. This is fine for multi-line expressions, but it's very annoying for simple single-line expressions. If you do a lot of simple one-line commands this becomes very annoying. And this is likely; few people type in long, multi-line definitions into a read-eval loop, because they would stick those into a file instead!

So how can we make indenting easy to use at a command line? We *could* create a special command for use at the beginning of a line, such as "! ", that means "at the end of the line, you're done" . Or we could create a special indicator that means "execute now" . But these don't seem any easier than entering a blank line, and they are yet something else to remember.

One simple modification could be to add a special case, based on whether or not text was entered in column 1. There are several obvious variants:

1. If leftmost edge has a "(" , and the *entire expression* is immediately followed by a newline, then the expression is complete. This has lots of problems; this doesn't handle evaluating atoms, such as running "x" to see what x contains. So I'll reject that.
2. If text begins at the leftmost edge (no leading horizontal space), and the following term is completely read, followed immediately by a newline, then the expression is complete and the read function returns. Thus, if these are entered at the leftmost edge, they will immediately return:

```
x
(load "file")
load("file")
(3 + 4)
```

However, the following will not (they will wait for a blank line) if entered on the leftmost edge:

```
load "file"
define x
3 + 4
```

The system will wait for a blank line in these cases, because the first term is followed by a space, not a newline... so it is waiting to see if you'll add more parameters by indenting them. The 'define x' is an example of good news (probably), and the 'load' command is an example of the bad news. Another problem is that the rule is a little complicated to explain. The good news is that 'define x' on the left edge will not trigger an immediate return. With these semantics, if you want to enter an indented expression, you'll need to indent the first line *or* include at least one parameter (expression) on the same line. If you indent at all on the command line, or enter one or more terms on one line, you must type an extra blank line to execute the sequence; this seems reasonable.

3. If text begins at the leftmost edge (no leading horizontal space), and there are no unmatched parentheses, it is *immediately* completed at newline. In this approach, you *must* indent the first line to have an indented expression. This means that if these were entered on the left edge, they would be executed immediately:

```
load "file"
3 + 4
define x
```

The good news with this approach is that 'load "file"' on the left edge works as a user might expect - it will trigger an immediate return. The same is true for 3 + 4 ... suddenly an interactive Lisp interpreter makes a plausible calculator!

The bad news is that 'define x' on the left edge followed by return will *also* trigger an immediate return of the read expression, which is almost certainly not what was intended. That is not such a big deal in an interactive command loop, since the user could try again. But more seriously, this would be a likely thing to do in a file, and it could cause very hard-to-detect bugs.

How can we counter the risks of not noting the change in semantics for text on the left edge? It might be possible to have a special loop for file-reading that detected the case of "unindented expression followed immediately by indented expression", and trigger a warning or error. Another variant would be to have two slightly different modes for read (or two different starting functions): one intended for interactive use, where unindented lines are ended on newline where possible, and one intended for file reading (where there is no such distinction). This does complicate reasoning and implementation, and it's quite normal to cut-and-paste into an interactive session.

One promising approach to countering the problem is to configure other tools to help detect such problems. Text editors, for example, could be colorize such constructs. Thus, if a left edge that would run immediately is followed by an indented line, it would be labelled with red lines. And special tools could be designed to detect this as well.

One good point is that the rule is simple: "if you want to indent, you must start by indenting the first line". When you indent, you enter a blank line or another expression at the same indentation level. Frankly, that's much simpler than the rules above.

4. If text begins at the leftmost edge (no leading horizontal space), and there is exactly one term on the line *or* it is a legal infix expression, then the

expression is complete and the read function returns. Thus, if these are entered at the leftmost edge, they will immediately return:

```
x
(load "file")
load("file")
(3 + 4)
3 + 4
3 + (4 * 2)
```

And these will not:

```
define x
load "file"
3 + 4 * 2
3 + 4 +
```

For sweet-expressions, for a long time I keep going back and forth between option 2 and option 3. Option 3 presents the risk of subtle bugs, and that is of great concern. However option 3 is a joy to work with interactively. I then said, "This is a hard choice to make, so I plan to experiment. Expressions like load("filename") and (3 + 4) seem particularly easy to explain, and do not have the drawbacks of this alternative, so they suggest using alternative 2." It was only in November 2007 that I finally realized that what I wanted was option 4.

Note that when activated these also partly eliminate one of the minor incompatibilities of I-expressions with traditional s-expressions. If there are two separate outermost s-expressions on consecutive lines, with the second expression indented from the first, as long as the first expression was not indented (a likely case) the lines will no longer be combined by the I-expression reader. Of course, this could be misleading, because the second line might *appear* to be combined with the first; this is a trade-off with no perfect answer.

Ignoring excess blank lines

Excess blank lines get interpreted oddly, as '(), which the underlying system may then try to run (Scheme in particular complains about them). It can also cause confusion if interpretation gets "out of sync".

A plausible modification would be to ignore any additional blank lines before a (next) expression. In short, a blank line shouldn't *really* return '() to the system; if they want that, users should need to enter that directly. If there blank lines at the end of the file, the system should probably return the end-of-file marker (if it supports that) once it has consumed the final blank lines trying to read the next expression.

An alternative would be to ignore any additional blank lines after an expression, but you don't want to do that. Then you need to treat the beginning of a file specially (assuming that there was a beginning of file).

Also, should horizontal spaces followed by newline be treated the same as a blank line? Users cannot easily see the difference, especially on typical printouts. That isn't clear, so I leave that as a question for now.

Indentation, a wrap-up

I-expressions and their extensions are a real improvement. But they are not enough; let's keep looking.

Lisp's standard notation is different from "normal" notation in that the parentheses *precede* the function name, rather than follow it. Jorgen 'forcer' Schaefer argues that this is a more serious problem than the lack of infix notation (http://sourceware.org/ml/guile/2000-07/msg00155.html); on July 2000 he said "I think most people would like Scheme a lot better if they could say lambda (expression) … instead of (lambda (expression) …"

Peter Norvig had some interesting ideas, as noted earlier. Let’s look at one of his rules, the rule that says “if a function name ends with an open parentheses, move it inside the list (when converting to an s-expression)”. This means that “(fact x)” and “fact(x)” will mean the same thing.

Obviously, this is trivial to parse. We don’t lose any power, because this is completely optional -- we only use it when we want to, and we can switch back to the traditional s-expression notation if we want to. It’s trivially quoted.. if you quote a symbol followed by “(”, just keep going until its matching “)” -- essentially the same rule as before! Technically, this *is* a change from some official Lisp s-expression notations and implementations. For example, entering “a(b)” into CLisp (a Common Lisp implementation) is the same as “a (b)” -- its parser tries to return the value of a, followed by running the function b. But it’s not clear it’s a big change in practice; commonly accepted style (http://www.lisp.org/table/style.htm) always separates parameters (including the first function call name) with whitespace. So normally, what follows a function call’s name is whitespace or “)”, and this is enforced by pretty-printers. Thus, many large existing Lisp programs could go through this kind of parsing without resulting in a change in meaning!

Does this help? Let’s rewrite the CL factorial example, but not make the infix operations do this:

```
defun(factorial (n)
  if((<= n 1)
      1
      (* n factorial((- n 1)))))
```

It looks slightly more familiar, but not that much. Let’s try again, but move the infix ops out too; the result is actually not bad:

```
defun(factorial (n)
  if(<=(n 1)
      1
      *(n factorial(-(n 1)))))
```

If we *really* wanted it to look conventional, we could use wordy names instead of symbols that are traditionally infix; that isn't too horrible for + (use "sum"), but that is rather wordy for others (I don't like it):

```
defun(factorial (n)
  if(lessequal(n 1)
      1
      product(n factorial(subtract(n 1)))))
```

Note that as far as I can tell, you do *not* need any whitespace after the opening parentheses, because atoms (including function names) cannot have parentheses in their name in s-expressions. A variant of this idea would be to require whitespace after the opening paren if name-prefixing is used, but I see no need for that.

Skill (http://portal.acm.org/ft_gateway.cfm?id=174185&type=pdf&coll=portal&dl=ACM&CFID=72577012&CFTOKEN=50239143) from Cadence, a proprietary Lisp-based extension language, also supports name-prefixing.

Mathematica does something similar. As noted in How Purely Nested Notation Limits The Language's Utility (http://xahlee.org/UnixResource_dir/writ/notations.html), its FullForm notation can be transformed into Lisp using simple rules, starting with transforming f[a,b] into (f a b). But it isn't designed to be backward-compatible with existing Lisp notations; its use of "," to separate arguments would cause confusion since "," is also used in macro handling.

Name-prefixing can be combined with indentation (e.g., I-expressions). Let's presume that if you indent further *and* begin the expression with a parentheses, or by a function name, no space, and a parenthesis, that parenthesis and its mate are silently ignored (they don't add yet another subexpression in the s-expression). Furthermore, let's presume that if you have a name-ender format (function name followed immediately by open paren), it does not switch to some "no more I-expression" mode. Then you could do this (using English names for operators):

```
defun factorial (n)
  if lessequal(n 1)
      1
      product n factorial(subtract(n 1))
```

Or this (using traditional symbols for operators):

```
defun factorial (n)
  if <=(n 1)
      1
      *(n factorial(-(n 1)))
```

It turns out that name-prefixing works well with indentation. Here are some examples:

```
; Here is (g (h 1)) <=> g(h(1))
  g
    h(1)
; or
  g h(1)

; Here is (g (h)) <=> g(h())
  g
    h()
; or
  g h()

; Here is (g (a) (b 1) c): <=> g(a() b(1) c)
  g
    a()
    b(1)
    c
; or
  g a()
    b(1)
    c

; Here is (g a (b 1) c()) <=> g(a b(1) c())
  g
    a
    b(1)
    c()
; or
  g a b(1)
    c()
```

This does introduce the risk of someone inserting a space between the function name and the opening "(". But whitespace is already significant as a parameter separator, so this is consistent with how the system works anyway… this is not really a change at all.

I think this is slightly better for untrained eyes. It's hard to argue that this is a major improvement, at least by itself, but the more I look at it the more I like it. What's more, the pain is tiny, and that's a good thing. This is a lower pain, lower gain approach, and it can be combined with SWP-expressions, which I'll describe in a moment.

The article Improving lisp syntax is harder than it looks (http://pschombe.wordpress.com/2006/03/17/improving-lisp-syntax-is-harder-than-it-looks/) discusses name-prefixing, but I think it makes a number of errors. It first claims that this would be hard to integrate with macros - this actually isn't true if it's built into the reader (and the macros aren't doing reading themselves). If the reader transforms a(x) into (a x), then when the macro has a chance to run all it sees is (a x) - exactly what it was expecting to see. He also says that "Another disadvantage to this change of syntax is that it makes functional programming much more odd looking. Lets say you have a list containing functions and you want to call the first one. In Scheme you write ((car lst) params) and in Common Lisp (funcall (car lst) params). However in our new syntax it looks like: car(lst)(params) and funcall(car(lst) (params)). Neither of these is very elegant, and it only gets worse if that call in turn returns a function, which would look like: car(lst)(params)(params2) and funcall(funcall(car(lst) (params)) (params2))." But I find this remarkably elegant, and *better* than the traditional notation - to do functional programming, just cuddle up the parentheses. It's *much* easier to understand sequential parentheses compared to a deeply nested list.

Again, let’s continue the thought that we want to support a syntax that is maximally Lisp-like, generally accepting existing expressions. The biggest issue in making Lisp s-expressions easier to read are in whether or not to provide infix support, and if so, how.

The first question is, should you support infix at all? Because if we do, and we presume that the underlying s-expressions always have the operation first, this means that we are *changing* the external presentation of (some) data. Note that we are *only* changing the presentation for entering programs and program-like data, and possibly changing how they are externally displayed; the traditional s-expression would still be used internally. Thus, if “{3 + 4}” is interpreted as an infix expression, it will quietly be transformed to the s-expression “(+ 3 4)”, so the “car” (head) of “{3 + 4}” would be “+”. The ll1-discuss mailing list (http://people.csail.mit.edu/gregs/ll1-discuss-archive-html/threads.html#01641) includes discussions on the issues of infix.

The reality is that nearly everyone prefers infix notation; people will specifically avoid Lisp-based systems solely because they lack infix support built-in. Even Paul Graham, a well-known Lisp advocate, (http://paulgraham.com/popular.html) admits that "Sometimes infix syntax is easier to read. This is especially true for math expressions. I've used Lisp my whole programming life and I still don't find prefix math expressions natural." Paul Prescod (http://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg01571.html) remarked, “[Regarding] infix versus prefix… I have more faith that you could convince the world to use esperanto than prefix notation.” Nearly all developers prefer to read infix for many operations. I believe Lisp-based systems have often been specifically ignored even where they were generally the best tool for the job, solely because there was no built-in support for infix operations. After all, if language creators can’t be bothered to support the standard notation for mathematical operations, then clearly it isn’t very powerful (as far as they are concerned). So let’s see some ways we can support infix, yet with minimal changes to s-expression notation.

If we are willing to support infix, there are many options. The big issues are:

1. How do you determine when to use infix notation? Is this manually specified (if so, how, and how deep does it go?) - or is it automatic (if so, how, and how do you override it?).
2. What are the legal infix operators?
3. What are the semantics? (Do you do something trivial, like swap the first and second parameters, allow many parameters for the same operator, or go all the way to full infix with precedence?)

For all of these, you also need to determine how to display them too.

There are some code samples that might be used as a starting point for implementations; after going over the issues, I identify some of them.

Nathan Baum (of Bournemouth, United Kingdom) posted on 4/14/2006 8:19:16 PM (http://www.gamedev.net/community/forums/topic.asp?whichpage=7&pagesize=25&topic_id=383805) the idea of using [...] to surround infix notation (I'm sure he's neither the first nor last). He said, "Lisp actually has a few syntactic shortcuts of its own. Essentially anything that isn't part of a name is a character which triggers a specialised reader function. Strings, for example, don't need to be built in to the Lisp core: a reader can be associated with the double quote character, and that'll return a string object. This also means you can redefine it yourself to get shell/Perl/Ruby-style interpolated strings, for example. Even lists themselves are read using a reader function triggered by [parentheses]. One simple shortcut is to have 'normal' Lisp expressions delimited by [parentheses], and infix expressions delimited by brackets...

```
(print (+ x (- (/ y z) a)))
; or
(print [x + [[y / x] - a]] ".
```

How do you determine when to use infix notation?

There are several major options:

- **Not automatic (special syntax)**. If infix is not automatic, you add characters that say "make this infix" , using pairs such as {...} or [...] around infix operators, or use single expressions before the first or second parameter. The single expression might be single characters, like ? or !, or multiple characters, like the dispatching macro characters (#...). I don't recommend "!"

because it looks too much like “not” in other langauges. Using something short is a good idea, because infix is very common. The Lisp FAQ (http://www.faqs.org/faqs/lisp-faq/part2/) mentions #I being used as the “Portable Infix Package” marker (I have searched and not found out more about this; suggestions would be appreciated). (This discussion (http://discuss.fogcreek.com/joelonsoftware4/default.asp?cmd=show&ixPost=141120&ixReplies=25) mentions an author using #I(3+4) to notate infix, and remarks that [3+4] would have been simpler.) *Requiring* a syntax marker is not as “pretty” as automatically notating infix operators, though it has the advantage of clarity if you process the resulting s-expressions often. is an example of this approach.

Pairs of characters can force infix interpretation, and seem like the obvious idea, but they do have some disadvantages. Infix operations can work across a list, so it makes sense to mark the end of such a list and use a special pair of characters like {...} instead of (...) to mark an “infix list”. There are two pairs available: {...} and [...]. Advantages of {...} is that they are more commonly used for statement blocks in other programming languages (Logo being an exception), and in particular users may want [...] for other purposes (such as identifying lists, like BitC). A disadvantage of using {...} is that in many fonts these characters look extremely similar to (...), making them harder to see; the characters [...] are more distinct. But in either case ({...} or [...]), using a pair means you have to match the correct closing pair, so inside many parens you have to make sure you use the right ones in the right place. This can create much extra work, and it is debatable if they help much in error detection. Also, infix notation is really obvious when you see it... so there really isn’t a big need for a special marker to show its end. Combining this with name-prefixing does not seem hard; just accept “{” for name-prefixing as well.

Combining the paired characters {...} or [...] with indented forms like I-expressions requires careful thinking about what you are doing. In particular, if you want to identify something as a parameter of a function f *and* switch to infix notation, do you indent inside function f? I think you should, so the system should only add *one* level of s-expression (from the indent) if the first character after the indenting whitespace is "{", instead of the two that would otherwise be implied. In all other cases, other than the beginning of a line, a "{" would normally end up matching a "(" in the corresponding s-expression if you want to stay similar to usual s-expression syntax (if not, see the relaxed infix notation such as ZY-expressions, below). This means that you need to think carefully about the meaning of {...} (or [...]). Basically, don't think that {...} create a matching s-expression (...). Instead, have the mindset that the *infix operators* (+, -, etc.) create the s-expressions, and that the surrounding characters {...} simply clarify that what's inside has an infix interpretation. Also, do {...} disable I-expressions inside them (as I've proposed (...) do), or should a "{" at the beginning of a line only switch to infix without disabling I-expressions? I'm not sure what the *meaning* of indented sub-expressions would even be, if indentation still mattered. Thus, for the moment, I'll assume that {...} disable I-expressions inside them, as I've proposed (...) do, for the simple reason of simplicity: if an infix expression is so complicated that it goes beyond one line, indentation like I-expressions are more likely to harm than help.

Note that if it's not automatic, there's a follow-on question... how deep does infix go when it is enabled? Do the infix operators only work at one level (one list), or do they keep going down to lists contained inside? This question doesn't come up if the detection is automatic, so let's address that next.

- **Automatic**. To work automatically, if some parameters look infix, then an infix interpretation is used. E.G., if presented with "(3 * 4)", it is automatically interpreted as “(* 3 4)” internally.

There are two major variants that I can see: (1) if the first *or* second parameter is an infix-type operator switch to infix notation (Norvig’s approach), or (2) only the second parameter can be infix to trigger infix processing (my approach). The first approach has the advantage that it can work with prefixed “-” traditionally, e.g., (- x * y) works, becoming (* (- x) y). But there are also many problems with that approach. For example, then it is *much* harder to use traditional Lisp notation (with the infix operator in front) at the same time, and it’s easier to get confused when looking at traditional s-expression notation. It is all too easy to accidentally trigger the infix notation when it wasn’t intended. And the traditional Lisp notation (- x) is not that hard to write or read; if you support traditional function notation, like -(x), it's even easier. Finally, if you *only* trigger on the second parameter, you can add a subtle safety check: the switch to infix notation can be made to *only* occur if there are three or more parameters. This safety check avoids unintentionally enabling infix in some cases, and makes it very simple to implement trivial quoting mechanisms: since expressions like “(self +)” will not be interpreted as being infix, using functions as parameters is simplified. In fact, you can go further on the safety check: You can say that infix lists have to have an odd number of parameters, and that the first parameter (as displayed) *cannot* be an infix operator (so a list of infix operators won't cause problems). I think the second approach is much better. For fully-traditional infix notation, the first is slightly better, and we’ll discuss that later, but for our purposes I don't think it's as useful.

If it’s automatic, now you need to determine the legal pattern of infix operators, since they will trigger the infix interpretation. This is probably some sort of regular expression; we’ll discuss this below.

You’ll also need a syntax to say when it is *not* infix at least (to disable the automatic part). Again, this can be identified using surrounding pairs such as {…} or […] around non-infix operators, or using single expressions before the first or second parameter to declare that this is *not* infix. For functions, in many real-world use cases Common Lisp’s #’ notation or simply “(function +)” would do. Indeed, if at least three parameters are required for infix, any function that simply returns its second parameter could do the escaping. As noted below, I'm currently using "as(+)" aka "(as +)" to do this escaping.

Automatic approaches, either way, are in some sense a little “hackish”.. the second parameter determines the order of parameters in the resulting s-expression! Yet it produces very nice-looking results, and I suspect that over time it will become workable. After all, in all other languages you have to look for infix operators too, so this actually not that strange a requirement. And the results (shown elsewhere) are startlingly clear.

You might still have an optional “make this infix” syntax indicator as well, or at least a “warning, this is infix”, which can be used when printing. That way, when you *read* a result as infix, you’ll get warned that the expression is infix. If this is optional, you might still want to make it very short, since you may have to read a lot of them in printouts. For example, let’s say that #_ means “no infix” and is placed before the first parameter (mnemomic: “base case”), while ? means “infix notation follows” is placed before the first (displayed) parameter (#^ might make a reasonable alternative). Thus, “(3 + 4)” , “(?3 + 4)”, and “(#_+ 3 4)” could all mean the same as “(+ 3 4”), and the “car” of all of them is “+” . Or, if […] are the infix warnings, [x >= 4] is a way to hint that the s-expression is actually (>= x 4).

Of course, if the notation permits mixed infix and prefix notations, you can easily have a “manual” notation that just means “turn on autodetection all the way down from here” and end up having a combination of properties. Alan Manuel K. Gloria’s “infix notation macro” is a good example of a mixture - in this implementation, there must be an initial marker (nfx …), but it then descends all the way down to all subexpressions. As a result, you could use the marker at a very high level, and have a result similar to automatic detection. A similar result could be had by having a function like “(enable-infix)” or “(sugar)” which you could insert at the beginning of a file as a top-level command (this could also enable indenting and the name-prefixing notation), which would then enable the infix operations (even at the topmost level) until specifically disabled. Using such a command would mean that everything below, for the rest of the file, would use automatic infix detection… but since there would be a specific a command to enable it first, it would be compatible with existing code (which didn’t enable infix). A command-line flag, or invoking a command with a different name (possibly via a different file extension), could enable automatic detection of infix notation.

What are the legal infix operators?

If detection of infix operators is automatic, now you need to determine the legal pattern of infix operators, since they will trigger the infix interpretation. Even if it’s manually triggered, if you permit arbitrary expressions, or want error-checking of the operators (”you said this was infix but there were none”) you’ll need to detect infix operators.

There are several options:

- You could support only a fixed list (+, -, etc.). That is very inflexible, though.
- You could start with a fixed list, and allow the user to specify which ones to add/override. The additions would probably include information on

precedence and direction (right or left), and if so, these additions would need to be made before they are used. One big risk is that if the infix expressions are read before the commands to make the additions, the infix expressions would be misinterpreted.

- You could define some sort of pattern. The pattern needs to be simple, so humans can easily remember it. This has the advantage of not depending on proper setup of a table of operators.
- You could support default patterns *and* allow additions as well, if you wanted to. Or define a pattern, but it's an error to USE the pattern before declaring it (and maybe its precedence).

If we use a pattern, what is the pattern? Patterns are often expressed as regular expressions (REs), and an obvious RE for this purpose is: [+-*/<>=]+. Basically, it has to be a set of one or more special characters. (One alternative, which might be nice, would be that it has to *start* with a special character, but after that, other characters are allowed; be wary that +h1 is an infix operator but +1 is not). This is enough to cover the 4 arithmetic expressions (+, -, *, and /), and all the usual comparisons (<, <=, =, >=, >). It's also enough to cover other operations that people like to have as infix, such as "**" (exponentiate), "->" (implies), "<->" (if and only if), and so on. You can probably add "&", which is enough to represent common representations of "and". But there are some issues, especially with the characters not noted here:

- Adding ":" is tempting, because ":" and "::" are sometimes used as infix operators (BitC uses ":" for declaring types). On the other hand, ":" is sometimes used as filler, so there are pluses and minuses to including it in the pattern. In Common Lisp, a ":" by itself is not a problem (it's illegal normally), but the ":" separates package names from their symbols. In addition, in Common Lisp a blank package name (e.g., :x) is a keyword… so anything with a colon at the beginning or middle is probably going to be a problem.

Similarly, a Scheme proposal (SRFI 88) (http://srfi.schemers.org/srfi-88/srfi-88.html) proposed using names ending in ":" as keywords, with SFRI 89 (http://srfi.schemers.org/srfi-89/srfi-89.html) using them for optional parameters. Thus, while a single colon (:) appears to be a fine name for an infix operator, more complex names with colons in them appear to be more problematic for backward-compatibility with existing Lisp-like systems.

- One possible concession would be to say that "and" and "or" are specially recognized as infix operators; that's a kludge, though *all* Lisp-like languages have "and" and "or" operators, so it would make sense. On the other hand, if infix operators automatically switch lists to infix notation, s-expressions of sentences like "(Jack and Jill)" would silently become "(and Jack Jill)" - and that seems dangerous. It especially seems dangerous if s-expressions are sometimes displayed with automatic infix notation without any indication that the parameter order has changed. So I think this is an unwise choice, even though at first it seems reasonable… but that still begs the question of how to represent "and" and "or", since these are incredibly common infix operations.
- Many languages use && or & to represent "and", and use || or | to represent "or". Adding "|" or "||" is tempting because it is a common notation for "or", and in Scheme the character is reserved for possible future extensions anyway. But simply adding it to the list of allowed symbols is more problematic, because in Common Lisp the "|" already has a meaning - it escapes symbol text until the next "|". So if "|" keeps its usual meaning, then "|" as an operator has to be written as "\|" in Common Lisp (yuk). But there's a clever trick here - the syntax "||" can always be used to mean "or", whether or not "|" has the special role of escaping symbol text. This would mean that in Common Lisp, the function's name would actually be an empty string (!)… but since its name would print as "||", it would *look* quite clear and understandable. This wouldn't be combinable with "=", (e.g., "+=" is a fine atom, but "||=" would immediately simplify to "=")… but that seems liveable.

It would be possible to force a literal | as the real symbol, but this would look ugly; \| or \|\| is hideous for code reading. In Scheme and many other LISPs this would be a non-issue; the || would just be interpreted as a two-character name for an atom, which is fine. The & can be confusing in some contexts, too; it is sometimes interpreted specially in parameter lists. But as far as I can tell, & is not normally interpreted in any special way outside of parameter lists, and even there, && isn't normally accepted. It's certainly possible to use "^" to represent "and" (using ** for the power operator), but that still begs how to represent "or"; few keyboards provide the mathematical "or" symbol (an upside-down wedge). So && and || are probably reasonable text representations for logical "and" and "or"... which is certainly consistent with typical practice. Note that "and" and "or" are special forms in Scheme (and many other Lisps), so that they can short-circuit; this means that redefining them is often more complicated.

- An unfortunate oddity happens with Scheme: Scheme actually defines a “=gt;” syntactic marker in its “cond” processing, and it occurs *as the second parameter*. This means that if you are using the second parameter to automatically detect infix operations, you will almost certainly consider this construct to be an infix operation. Unfortunately, you cannot solve this by creating a new simple function => that can handle this (having only one form at this location changes the meaning of the construct). There are several solutions; two obvious ones are to require manual detection or to have a specific list of infix operations. Another solution is to specifically state that “=>” is excluded from the pattern of allowable infix operators, at least for Scheme. You can partly justify this on the additional grounds that -> and => are easily confused. It would probably be possible to solve this with a complex Scheme-unique macro for => that essentially restored its original meaning, though this will probably cause problems if an error occurs nearby; I think simply declaring that it's not an infix operator when processing Scheme is safer.

- How do you represent assignment and field accessors, if the underlying language has them? The pattern supports "=", but if you use "=", then you have the problem that this is easily confused with equal-to. This is one of the big problems with C and C++ (confusing = with ==), and it'd be unfortunate to duplicate that mistake. In addition, if the system maps any infix name directly to the identical prefix function names, then you may not have a choice... in most Lisp-like systems, "=" is already an equality operator. The term "<-" is compelling for assignment, and "->" is compelling for field access, but the two do not go together well; just imagine deciphering "x -> a <- b -> c". Parentheses help, e.g., "(x -> a) <- (b -> c)" - but it's still a little awkward.

  One possibility is to use a different form for assignment, such as "<--" or "<==". The term "<==" is actually fairly easy to distinguish, so "x -> a <== b -> c" is easier to read, and "(x -> a) <== (b -> c)" is actually especially nice.

  Another possibility is to use "<-" for assignment, and use [...] as a field accessor. That looks nice:

  ```
  x[a] <- b[c]
  ```

  But if we use [...], we need to figure out what general function to map [...] to, *and* we just lost the phrase [...] for other purposes (such as describing special lists). (For sweet-expressions, I'll presume that the language or user will define the infix macros/functions, and not add [...] simply because it's easier and more general not to.)
- Since Unicode/ISO 10646 is finally becoming widely available, and many systems can process them (e.g., using UTF-8 encoding), additional infix symbols could be accepted that are *not* in ASCII. This would certainly resolve the “and” and “or” issues, since there are standard mathematical symbols for them: ∧ ("logical and") is character U+2227 (decimal 8743) and ∨ ("logical or") is character U+2228 (decimal 8744). See Unicode symbols chart

(http://www.unicode.org/charts/symbols.html), especially the mathematical operators chart, for more information. The character set from U+2200 through U+22FF is allocated to mathematical symbols. This will remind those of us who've been around a while of APL! Probable characters for infix operators include U+2227 ("and"), U+2228 ("or"), U+220A ("element of"), U+2209 ("not an element of"), U+220B ("contains as member"), U+220C ("does not contain as member"), U+2229 ("intersection"), U+222A ("union"), U+225D ("defined as"), U+2260 ("not equal to"), U+2254 ("colon equals"), U+2282 ("subset of"), U+2283 ("superset of"), U+2284 ("not a subset of"), U+2285 ("not a superset of"), U+2286 ("subset of or equal to"), U+2287 ("superset of or equal to"), U+2288 ("neither a subset of or equal to"), U+2289 ("neither a superset of nor equal to "), U+228A ("subset of with not equal to"), U+228B ("superset of with not equal to"), U+22C8 ("bowtie" - sometimes used for join), U+22BB ("xor"), and U+22BC ("nand"). But not everyone has good support for these yet, so while include a set of Unicode infix operators in the standard set might be a good idea (anticipating their use), it's probably premature to *require* their use.

If the printing routines normally convert to infix, then you need to *not* have a pattern that is likely to be engaged unintentionally… which are good reasons to *not* have the terms "and" and "or" be infix operators, and certainly a good reason to stay away from anything other than an isolated ":". You can probably limit the regular expression length, say up to 3-6 characters, to help limit unintentional conversions as well. I would not include "and" and "or", instead add & and | to the infix operators (and say that | has to be escaped), and limit it so it must be 1 through 4 characters in length. I would accept ":" alone (so it can be used for infix type declarations), but nothing else with colons.

These considerations suggest that "=>" be quietly prevented from being an infix opreator, and then use this regular expression for infix operators;

```
    [+-\*/<>=&\|\p{Sm}]{1-4}|\:|\|\|
```

where “\|” means match the “|” character, and \p{Sm} means “match mathematical symbols, Unicode range U+2200 through U+22FF”.

As a related matter, you need to decide how to separate infix operators from non-infix operators. In many other language families, “x-y” would be parsed as x minus y. Unfortunately, in many Lisp-like languages, names often include typical infix operator characters such as “-”. Both Common Lisp and Scheme would be unusable without the - symbol, because they have many functions with “-” in the name (in addition, Scheme has a convention where “->” embedded in the name indicates a conversion). The simplest and most obvious approach is to require that any infix operator be surrounded by whitespace; this is very easy for humans to remember, and is consistent with normal s-expression syntax anyway. You *could* require escaping such characters as part of a name, e.g. |simple-string-p| or simple\-string\-p, but this is both ugly and unnecessarily incompatible with common practice. If you’re devising a new language, you could simply forbid such characters in regular symbol names, and then you could automatically detect all the infix operators without requiring that they be surrounded by whitespace. However, for general-purpose parsing, sweet-expressions will require that infix operators be surrounded by whitespace (in cases where function names don't have such symbols, this requirement could be relaxed).

What are the semantics of the infix operations?

Once we detect that we are using infix notation, what are the semantics of the “infix” notation?

1. **Swap first two parameters**. An *extremely* easy-to-implement and general approach is to just swap the first two parameters. A good idea for this case

would be to require exactly 3 parameters -- that way, we won't accidentally screw things up, because when limited to exactly 3 parameters, there's no difference between swapping and the normal interpretation of infix. This is trivial to implement, and yet it's enough to implement basic infix operators in a way that looks nicer. This means that the presented expression is very similar to the actual s-expression, which has its advantages. If you want to add a fancier infix format later, this approach is a reasonable stepping-stone towards that.

2. **Allow chaining (duplication) of identical operator**. This extends the previous option, but you can have an odd number of parameters, and the even ones much match. Thus, (3 + 4 + 5 + 6) becomes (+ 3 4 5 6). Trivial to implement (just swap the first two parameters and ensure the rest match). This approach makes it possible to fully use the capabilities of underlying functions that allow multiple parameters, and that's a good thing. Interestingly enough, Common Lisp's definition of comparison operators would make expressions like (x <= y < z) work as it does in mathematics!

Note that this does *not* support precedence, so you have to surround different operators with parenthesis or indents, e.g., (3 + (5 * 6)). In one view, that's a disadvantage - you still have to indicate some information that would be automatic in other languages. On the other hand, this makes *when* there are new lists explicit, and that has some pretty big advantages. It also sidesteps the problems of defining precedence (either fixed or a way to add them). Those are pretty big advantages.

There are many options for determining when things aren't correct, and then deciding if that is an error or merely an indicator that infix was not intended. Note that this only supports an odd number of parameters, and obviously there shouldn't be an infix operator presented as the first parameter in the text. Doing otherwise could be an error, or could be an indicator that it should not be treated as infix. For the moment, I'll plan to consider both as an indicator that infix was not intended. This only permits all the operators to be identical - if they're different, but all infix operators, then the writer probably intended some sort of precedence (and so an error should be reported). Otherwise, infix was probably not intended, so again, I'd probably interpret that as "not infix".

3. **Full infix of binary operations, with precedence rules**. Allow infix and add precedence rules (* before +, etc.). This would mean that "(3 + 4 * 5 ** 2)" would quietly transform into the s-expression "(+ 3 (* 4 (** 5 2)))". In cases where it is important that you be able to clearly see the mapping between the surface presentation and the underlying s-expression, just don't use precedence - instead, parenthesize (so those who do not need to see that representation don't have to). This is trivial to implement in Lisp… the problem is what to do about the precedence of unknown operators. You'd need to either not support precedence of them, have a default for unknown operators, or have a way to set precedence values (and make sure the settings are read before processing the code). Setting precedence is easy enough, but its disadvantage is that it invites trouble - you need to make sure that the rules are set *before* the parameters are swapped, and mistakes could result in hard to find errors. Left association could probably be safely assumed for more than one instance of the *same* unknown operator. Alternatives are giving unknown operators a specific precedence relative to other operators… or simply *requiring* that other operator's precedence be made clear with parentheses (or their equivalent). Predefining precedence for common functions, and requiring statements about the others, isn't a bad thing; people usually don't want to have varying precedence tables (it's confusing), and they also don't want to memorize large tables. Again, I suggest making duplicate operators (3 + 4 + 5) turn into one s-expression (+ 3 4 5); have the user override if they want something different.

Here, we don’t allow prefix/suffix operators, such as “- x”, or suffix operators, like “x !”. There is a trade-off here; prefix/suffix don’t work well with automatic detection or trying to also read traditional s-expressions correctly - it would be all too easy to misread such expressions. Using name-prefix notation for them, such as -(x), is a very reasonable alternative.

4. **Full infix of binary and non-binary operations, with precedence rules**. This is like the above, only now we also allow prefix operators, such as as “- x”, or suffix operators, like “x !”. Allowing them adds a little extra flexibility, and might make sense if the specific expression is always manually marked as being infix. But it has many problems, as noted earlier.

Precedence rules

If you have precedence rules, what are they, and are they controllable? It's not hard to add a command that sets precedence rules, but there's always the problem that you risk not getting that called at the "right time" (before reading). Worse, if there are global precedence rules that can be changed, setting them in one environment may manipulate another (unintended) environment, so now we have to have a parameter for (read), and that may be hard to pass down. This is particularly a problem for Lisp-like systems, where you may have multiple levels and meta-levels that you might not want to interact. Another big challenge is that if different developers set precedences, it's harder to combine their code. Different operations may have different meanings in different circumstances, so the lack of control over precedence can be a problem too.

A chain of the same operation can normally be combined into a single operation with all those parameters, e.g., (a + b + c) should become (+ a b c). This adds capability, without any problems or loss.

Everyone agrees that * and / should have the same precedence, both of which have greater precedence than the equal-precedence binary + and -, and that all of these are left-to-right. So in theory that, at least, could be implemented. Originally I thought that it may be best to just implement those universal rules, and normally use parentheses most everywhere else for controlling infix precedence (or at least discourage lots of precedence-setting). Chaining is fine, but as discussed below, precedence causes a lot of problems in many use cases, so in the end I decided against them for sweet-expressions.

Printing expressions that might be infix-able

So far, I've primarily discussed how to read in expressions, but expressions need to be printed too. A likely answer is "just print s-expressions as usual" by default. That means that the output would be "(+ 3 4)", even if the input was "(3 + 4)". For debugging code, this is probably a Good Thing. There might be value in presenting expressions back with some infix operations "re-inserted", at least as some sort of "pretty-printing" function or option. It might be wise to identify which lists are being interpreted as infix in these cases (e.g., by surrounding them with {...}). I think there is no reason to try to redo precedence when re-displaying; showing how the lists are actually stored is very clear and eliminates questions about precedence. Thus, if "(3 + 4 * 5)" is input, the resulting s-expression would be "(+ 3 (* 4 5))", and the infix-printer might show this as "{3 + {4 * 5}}".

Combining infix with Name-prefixing

There is a subtlety when combining name-prefixing with automatic infix: I think people expect an infix expression to be considered a *single* expression, even though at first it appears to be a multi-parameter list.

For example, most people would accept this syntax for calling function f with two parameters, x and y:

```
f(x y)
```

However, if infix is normally done automatically, they would expect this to be computed with f given one parameter, not three:

```
f(x + y)
```

Note that this is actually a very nice thing; it means that in many cases, name-prefixing causes the number of parentheses that need writing to reduce. So the original Lisp:

```
(f (+ x y))
```

becomes:

```
f(x + y)
```

Infix control

There is likely to be a need to control infix processing. These should be a standard way of saying “disable infix” and “enable infix”. You'd like to be able to enable or disable infix at only one level, in particular, so that you could leave infix on as the default and yet disable it in one particular expression. It would be especially useful to have an “enable infix for this one level of expression” operation that *must* receive an expression that “looks like an infix expression” or it is in error.

The following text are some very early ideas on the topic; I include them here, but it's all very early, and at least some is likely to be very wrong. Still, you may find useful thoughts here.

One implementation issue is that macros work from outside in, and most macros will not expect special additional macros. In particular, it might be very useful to cuddle an infix operator with a macro that says do not use infix, like this (where "as" means "as-is"):

```
(define as(+) ...)
```

However, the obvious solution will fail; typical implentations of "define" do not expect to be followed by another macro in the definition! This means that any such function reference will need to be removed at *read* time or very early on during *eval* time (e.g., by an outside cuddling nfx(...)), because other macros won't know what to do with this.

There should be a standard way of inserting at the beginning of a file or read-eval-print loop “please switch to and from special processing, and for setting some of its options (e.g., infix control). For debugging, you could just print s-expressions as now. However, a standard way to request printing these expressions that shows infix as such, but marking infix operators and non-infix, would be a good idea and each list which could be misinterpreted as an infix expression but is not has another marker. This implies a need for a marker that says “the following list *is* an infix expression and it’s an error if it is not”. If adding specific new infix operations is supported (such as “and” and “or”), a standard name and syntax for this add operation would be a good idea too. Ideally, the external interfaces for these operations be pseudo-standard across all Lisp-like notations. Bridging the gap between Common Lisp and Scheme might be challenging for a standard interface (in some cases string parameters might be needed; string syntax is now universal, but other syntax is not - e.g., the keyword syntax of Common Lisp is incompatible with the proposed keyword syntax for Scheme).

The notation for controlling all this is to be determined. Here's a start. Originally, I looked at implementing another # character option. If you want to do that, you might look for sharpsign options that no one one currently uses, examining sources such as the (particularly the sharpsign section (http://www.lisp.org/HyperSpec/Body/sec_2-4-8.html)), the R5RS Scheme specification (http://www.schemers.org/Documents/Standards/R5RS/HTML), the GNU Emacs Lisp Reference Manual (http://www.gnu.org/software/emacs/elisp-manual/), as well as less-common systems like NewLisp (http://www.newlisp.org/downloads/newlisp_manual.html). Using #I for infix is tempting, but Scheme uses #i for inexact; it's not clear if #I would get interpreted the same way on some systems, but it's not worth risking and might be confusing to developers anyway. I especially considered using #/ to mean "infix". The #@ combination is used by Emacs Lisp, so we should avoid that. However, Scheme only supports one-character lookahead; if you want to be able to call the "ordinary" Scheme reader for other #-beginning constructs, there isn't an easy way to do that - because to read the character after #, you have to consume the #, making that more difficult to do. So I don't think beginning with # is a good idea.

I've looked for some possible notation, under the presumption that this is a future standard format common to Common Lisp and Scheme (so we need something unallocated by either).

A simple approach might this naming convention, with function-call-like macros:

- nfx(...): Everything inside is interpreted as infix if it can, recursively.
- unfx(...): Everything inside is NOT infix, recursively.

- nfx1(...): The immediately contained inside is infix, and it is an error if it is not.
- unfx1(...): The immediately contained expression is *not* infix.

Each of the above could either accept a single list (in which case it's the list that is being referred to), otherwise it is the expression itself as the rest of its parameters that is being referred to. Since any infix expression must have *at least* three parameters (2 if unary operators are used), this isn't ambiguous.

We could interpret unfx(...) around a second parameter to quietly disable infix processing. Alternatively, we could have a different function/macro name such as "as(...)". This would Treat the inside (function) as-is, so it won't be considered an infix operation (use this if an infix operator is the second parameter, but you don't want the expression to be considered infix). Here's an example:

```
defun as(+) (left right) ...
```

One interesting challenge in doing this is recursion in the read function. I decided to implement much of the processing in the reader, and to have the processing go "as I go" in the reader, rather than have the reader automatically add nfx() in front of expressions and then have eval() fix it up by calling a macro. Since the reader is called on ordinary data in s-expression format, it is very inconvenient to automatically have the reader add nfx(), etc. calls! Yet if there is no outermost call to "fix up" the expression, then the outermost parameters would be defun, define, or other terms that would not handle these macro calls correctly. And you dare not remove things piecemeal - read can be called recursively, so if you call read, remove things partly, but then later all "fix up" all the way down again, you can "unfix" things. This is one reason to have a separate "as" function, so that these different forms can be differentiated.

Note that "(s (b))" is interpreted as you might expect during sweet-expression processing, but (s(b)) is interpreted as the s-expression ((s b)), not (s (b)). In practice this is not a problem - *nobody* likes the format (s(b)) for s-expressions, and not all s-expression processors would even accept them. More also needs to be done to describe their interaction with macros; this is a critical area and hard to get right, but since there is *relatively* little that is changing, this may not be so bad.

A simple program could transform an existing program to sweet-expression format, including its comments, and with any necessary markers to control infix interpretation. I expect that the need for controlling infix interpretation will be exceedingly rare, so the results should look very nice.

Infix implementations

Many, many people have implemented infix processors. Here are some, not including the many larger notational systems (like IACL2) that have an infix notation built into them:

1. A simple Lisp program that converts infix to s-expression format (http://www.cs.berkeley.edu/~russell/code/logic/algorithms/infix.lisp) is available as part of Peter Norvig's book, "Artificial Intelligence: A Modern Approach." (http://www.cs.berkeley.edu/~russell/code/doc/overview.html) This code's license (http://www.norvig.com/license.html) appears to be open source software. Besides the usual disclaimers, it says, "3. The origin of this software must not be misrepresented, either by explicit claim or by omission" and "4. Altered versions must be plainly marked as such, and must not be misrepresented as being the original software. Altered versions may be distributed in packages under other licenses (such as the GNU license)."
2. Alan Manuel K. Gloria's "infix notation macro" (http://plaza.ufl.edu/lavigne/infix.lisp) is extremely promising. His approach

is to create a macro “nfx”, and then put spaces around everything. From then on, everything *inside* transitively *can* use either infix or prefix notation. It detects which lists are infix by examining the second parameter, and determining if it is in a list of infix operators. At this time the license intends to allow arbitrary use, though not in a clear legal manner.
3. wrote an infix reader macro that is typical of many such packages (dated Jan 18, 1995); his runs on Common Lisp. It is often mentioned, though its restrictive license makes it useless to many. It “allows the user to type arithmetic expressions in the traditional way (e.g., 1+2) when writing Lisp programs instead of using the normal Lisp syntax (e.g., (+ 1 2)). It is not intended to be a full replacement for the normal Lisp syntax. If you want a more complete alternate syntax for Lisp, get a copy Apple’s MLisp or Pratt’s CGOL. Although similar in concept to the Symbolics infix reader (# <DIAMOND>), no real effort has been made to ensure compatibility beyond coverage of at least the same set of basic arithmetic operators. There are several differences in the syntax beyond just the choice of #I as the macro character. (Our syntax is a little bit more C-like than the Symbolics macro in addition to some more subtle differences.) It is not open source software; derivatives are allowed, but no fees or compensation may be charged for “use, copies, distribution or access to this software”, which probably forbids Linux distributors from distributing it (it is in Debian’s “non-free” section for this reason). Here it is (http://www-cgi.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/code/syntax/infix/). Norvig extended it (http://www.cs.technion.ac.il/~shaulm/software/infix.lisp). With this, you can write #I(3+4). It supports all sorts of precedence rules, etc. However, its restrictive license makes it useless to most people; it cannot be distributed by many FLOSS vendors, it cannot be used in commercial settings, and since there is no hope of commercial support for it, its code is unusable to most people. However, it is a good idea that with a fresh re-implementation might be practical for some (after all, this package is a re-implementation itself).

4. ”mlish” (http://www.accesscom.com/~darius/) is an infix syntax frontend for Scheme (from the same person who made “indent”).
5. The Gambit Scheme implementation (http://www.iro.umontreal.ca/~gambit/doc/gambit-c_17.html) reader includes the SIX (Scheme Infix eXtension). “The backslash character is a delimiter that marks the beginning of a single datum expressed in the infix syntax … the backslash character escapes the prefix syntax temporarily to use the infix syntax. For example a three element list could be written as ‘(X \Y Z)’, the elements X and Z are expressed using the normal prefix syntax and Y is expressed using the infix syntax. When the reader encounters an infix datum, it constructs a syntax tree for that particular datum. Each node of this tree is represented with a list whose first element is a symbol indicating the type of node. For example, ‘(six.identifier abc)’ is the representation of the infix identifier ‘abc’ and ‘(six.index (six.identifier abc) (six.identifier i))’ is the representation of the infix datum ‘abc[i];’.
6. guile-arith (http://www.cl.cam.ac.uk/users/ig206/guile-arith/) is a module to demonstrate how to build a parser in the Guilde version of Scheme, but is very immature. Its web page says “this is not meant to be used.” It builds yacc/lex parsers into guile C modules. It has a demo, supporting notations like this:

```
(define (square x) #[ x ^ 2 ])
(square 2) ==> 4
(define (cube x) #[ square(x) * x ])
(cube 2) ==> 8
```

7. Note that SIOD (Scheme in One Defun) (http://www.cs.indiana.edu/scheme-repository/imp/siod.html) also included an infix module - again, not something that is standard across implementations.

Putting it together: Sweet-expressions

Obviously, there are *lots* of different ways to try to make Lisp-like languages easier to read. So let’s combine them. In particular, I’ve identified a particular set of options that I think are tasty.

Sweet-expressions: A basic definition

I have developed a combination I call “sweet-expressions” that I hope will be a good solution. In this notation, normally-styled s-expressions will typically work “as normal”, but various extensions are added that make it possible to create easier-to-read programs:

1. We’ll support indentations using augmented I-expressions. An indented parameter with no following parameters (on the same line or via indentation) is considered a symbol, so a function with no parameters must be invoked by surrounding it with “(...)” or terminating its name with “()”. We'll augment it by ignoring blank lines at the beginning of an expression, and by immediately executing a term that begins at the left edge and is immediately followed by newline (to make interactive use pleasant).
2. By default, any expression is an infix expression if the first term (the function name) is *not* an infix operator, its second term *is* an infix operator, and it has at least three terms; otherwise it is a traditional prefix expression. Infix expressions must have an odd number of terms, and the even parameters must be the identical infix operators. So, (2 + 3 + 4) is okay, but (2 + 3 * 4) is not; you must use (2 + (3 * 4)). That way, you *only* need to look at the first two terms of any expression to see if something is infix or not. If all the (odd-numbered) infix operators are the same, they are chained into a single infix operator, so (2 + 3 + 4) becomes (+ 2 3 4). Note that in a name-prefix form, it's the parameters that are considered, so f(2 + 3) becomes (f (+ 2 3)). Lists

where the even-numbered parameters are different are considered non-infix (at least if any of them aren't operators); that will mean that more "traditional" Lisp s-expressions will be read without meaning change, even if they have an operator in the second position. You must separate each infix operator with whitespace on both sides.

Previously I planned to use built-in precedence rules if the infix operators differ, but after some experience I'm currently thinking that it'd be better to *not* support precedence at all. Precedence is nearly universal among programming languages; they're very useful, and only a few infix-supporting languages (such as Smalltalk) lack them. But in the typical use cases of a Lisp-like language, precedence has some significant downsides that are irrelevant in other languages. First, let's talk about a big advantage to *not* supporting precedence in sweet-expressions: It makes the creation of every new list obvious in the text. That's very valuable in a list processing language; the key advantage of list processing languages is that you can process programs like data, and data like programs, in a very fluid way, so having clear markers of new lists like parentheses and indentation is very valuable.

What's surprising is that precedence turns out to have a host of problems in specific context, leading me to believe that it's a bad idea for this usage. Precedence is very useful in general (and it could be added for special cases), but adding precedence rules to a general-purpose list expression processor creates a slippery slope of complexity. You can trivially create commands that create new infix operators, and express their precedence and direction - no problem. But you're using a Lisp-like language, the use case tends to be different, with objects shifting between being data and being code. Often you're working at multiple levels and meta-levels of input - and it's not likely that you'll want their precedences to be the same. Yet trying to handle that turns out to require very complex management of the reader routine. Also, combining code from different sources can be a nightmare if they have different precedence values. In short, having precedences vary by meta-level creates complexities you just don't want to deal with. That suggests moving to a completely static, pre-canned set of static levels is required. Okay, but someone has to predefine that set. That turns out to be really hard; the meanings of various infix operators are *not* cast in stone, so it's difficult to preset the precedence levels meaningfully (since you don't know what the operators mean). The one obvious case is that * and / have higher precedence than binary + and - basically everywhere. That particular rule *could* be built-in, but if that's *all* you build in, it's not clear that it's worth the trouble. Especially since the idea that completely forbidding mixing the operators means that we make the mapping between lists and the textual representation much clearer. It's easier to add precedence later, if that turns out to be important after more experimentation. But after the current experimentation it appears that precedence simply isn't worth it in this case; it creates complexity, and hides where the lists begin/end. Precedence is great in many circumstances, but not in this one.

Of course, you can write code in some Lisp dialect to implement a language that includes precedence. Many programs written in Lisp, including PVS and Maxima, do just that. But when you're implementing another language, you know what the operators are, and you're probably implementing other syntactic sugar too, so adding precedence is a non-problem. But sweet-expressions are intended to be a universal representation, just like S-expressions are, so their role is different… and in that different role, precedence causes problems that don't typically show up in other uses. Not supporting precedence turns out to be better in this different role.

This approach supports binary infix operators, but doesn't include any special mechanism for unary minus. For unary minus, you must use the name-prefixed form “-(x)” or the prefix form “(- x)” . By only handling binary infix operators, it is much less likely for an expression to be “accidentally” interpreted as an infix expression when we did not mean it to. For example, you can even have quoted lists of infix operators, and freely mix s-expressions that begin with infix operations, without trouble. But there are a few cases where automatic infix interpretation while reading can be troublesome, so there will be ways to control it. Note that an expression is *not* infix if you do just about anything with the second parameter, or if you use an infix operator as the first operator. This means that:

```
(2 + (3 * 5)) <==> (2 + (* 3 5)) <==> (+ 2 (* 3 5))
```

3. We’ll allow the use of name-prefixing, so NAME(x y) is the same as (NAME x y). NAME must be followed *immediately* by a “(” without whitespace between them. For the infix rules purposes we’ll perform this transformation *after* doing infix expression handling, so if the cuddled expression is an infix expression, it's a single-parameter function with an expression in infix form (you can use parentheses to cause a multi-parameter interpretation). Here are some examples:

```
factorial(z)        <==> (factorial z)
foo(x y)            <==> (foo x y)
bar(x y z)          <==> (bar x y z)
factorial(x - 1)    <==> (factorial (x - 1)) <==> (factorial (- x 1))
f(g(y) h(y) a)      <==> (f (g y) (h y) a)
f((x + 2) y (z - 3)) <==> (f (+ x 2) y (- z 3))
f((x + 2) * (y - 3)) <==> (f (* (+ x 2) (- y 3)))
```

I call this combination “sweet-expressions” - by adding syntactic sugar (which are essentially abbreviations), I hope to create a sweet result. In the few cases where the sweet-expression is awkward, just use the usual s-expression notation (with infix escapes if necessary) -- well-formatted s-expressions are still legal. The result supports quoting and backquoting, is very compatible with typical s-expressions, and should work well with most macros.

Controlling sweet-expression interpretation and next steps

There should be standard ways controlling this, particularly for infix interpretation. Here are some early ideas about this.

For the moment, I plan to just implement an "as(..)" expression that can be cuddled around the infix operator (the second parameter). This must be removed at *read* time or very early on during *eval* time, because other macros won't know what to do with this. In other words, to ensure that "(defun as(+) ...)" will work correctly, we need to process as(..) *before* we process "defun". (Alternatively, use name-prefixing: defun(+ ...) works as well). I plan to implement this at read time for now, so that all other macros will work perfectly normally.. by the time normal macros get the expressions, they only see standard s-expression format. (Instead of as(...), I could use some notation beginning with #, but this turns out to be awkward in Scheme (because handling # as a secondary reader is painful), and it also turns out to be awkward in Common Lisp (because functions have a different name space, so you have chained # operators that look confusing).

Controlling infix processing is probably the most troublesome issue. For new code, it's more convenient to presume infix unless disabled - you really only need the occasional "as". Yet for maximum compatibility with existing code, it's better to expressly turn it on and to have control over it, so you want default off, with various ways to enable it. I believe this is very solvable.

For now, I'll just implement the "as" version, which is enough to see how it works.

A related issue is defining what the infix operations are. It seems clear to me that = (equality), <=, +, and so on should map to their prefix versions. I also think that && should map to "and", and || should map to "or" -- these are common conventions in other languages, and thus should be easy to read. I think "**" should become "expt" (exponentiate). But there are many other prefix operations with text-based names that should be defined (by defining a function or macro) that aren't as obvious. All these mappings can be done using regular definitions, without anything special happening in the language.

First, assignment (setf in Common Lisp, set! in Scheme). Gloria suggests using "=" for setting values (setf in Common Lisp), but I think that's a bad idea. After all, "="... that's already the equality test, and having the same symbol mean different things is likely to be disastrous; C already has trouble simply because "=" and "==" look too similar. The phrase ":=" has its problems too; it'd be considered "=" in the keyword space (confusing!) in Common Lisp, and doesn't prefix well. A plausible alternative is "<-", so "x < 5" reads as "set x to the value 5". This also combines well with C-like operators, e.g., "+<-" could be "add to"... that means that "x +< 5" would be the same as "(setf (+ x 5))" in Common Lisp or "(set! x (+ x 5))" in Scheme.

There are lots of other potential operators. Gloria suggests "@" become prefix "aref" (which accesses array "left" at element "right"), "->" become "slot-value-q" (though this conflicts with using it as "implies"), "%" become "mod", "===" become eq, "==" become equal, "&" become logand, "|" become logor, and ":" become "values". I worry about using | and & because they're easily mistaken for the double-character kind. Indeed, Dennis Ritchie admits that the precedence values for bitwise and/or, vs. logical and/or, were a mistake (http://www.quut.com/c/dmr-on-o) - made worse because of their visual similarity. Languages should designed so that people will be less likely to make obvious mistakes (http://leshatton.org/index_SA.html), based on previous measured evidence that these mistakes occur... and this evidence certainly exists in the case of && vs. &.

What should ++ be... increment, string concatenation, something else? There should probably be a string concatenation infix operator, but I'm not sure what to name it.

I would use "->" for implies, and "<->" for iff (if and only if). One danger is that mixing "->" and "<-" in the same expression can be very confusing, but I don't see a simple alternative.

A "standard list" of common infix operations and their meaning would make it easier to read and understand such expressions. But this is difficult to do, and in case only makes sense once there's more experience with sweet-expressions.

What about writing a list back out? It'd be nice to have a written form that could always be read back in as a sweet-expression. An optional writer would be nice - one that added as(...) where needed, and which optionally showed infix as infix, or optionally used indenting. It could be made really obvious that it was the special writer by using name-prefixing where possible -- that would be a glaringly obvious difference from the traditional s-expression writer. So it would write stuff like:

```
if((x = 5) display("Five") display("Not five"))
```

Alternatively, the writer could normally write *without* using name-prefixing:

```
(if (x = 5) (display "Five") (display "Not five"))
```

It could also just write traditional s-expressions, but adding the as(...) in the few places where it's necessary; that'd be good for debugging:

```
(if (= x 5) (display "Five") (display "Not five"))
```

Simple Example

Now, let’s look again at our sample Common Lisp program, using traditional s-expression notation:

```
(defun factorial (n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))
```

The Scheme version is similar:

```
(define (factorial n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))
```

The two samples above are *also* legal sweet-expressions. Why? Well, the initial “(” disables I-expression processing, so even if the formatting is ruined, we’re simply processing as usual. There’s no function name ajoined to the right with “(”; normal formatting rules wouldn’t even consider that. As is usual for Lisp code, “infix” operators are listed first (not second) so there’s no infix operator as a second parameter to cause a change.

But what if we take advantage of the new capabilities of sweet-expressions? A pretty-printer could even do this automatically, as a way to start.. and the results look fantastic. Here's the Common Lisp version:

```
defun factorial (n)
  if (n <= 1)
      1
      n * factorial(n - 1)
```

And here's the Scheme version:

```
define factorial(n)
  if (n <= 1)
      1
      n * factorial(n - 1)
```

What’s even more interesting, we haven’t lost any power. We can still use quotes, or quasiquotes with comma-lifting. We can still have functions that return functions, and so on. We can also mix up regular s-expressions as part of sweet-expressions, or even mix things up back and forth:

```
defun factorial (n)
  if (n <= 1) 1 (* n (factorial (n - 1)))
```

Another example

Here's a trivial example derived from the Common Lisp Tutorial (http://www.notam02.no/internt/cm-sys/cm-2.2/doc/clt.html) (by Gordon, Haible, and Taube):

```
(do ((x 1 (+ x 1))
       (y 1 (* y 2)))
      ((> x 5) y)
    (print y))
```

Again, the above is also a valid sweet-expression, but we can make it look much nicer instead:

```
do
  group
    x 1 (x + 1)
    y 1 (y * 2)
  group
    x > 5
    y
  print y
```

Some patterns

More complex s-expression patterns turn out to have regular representations in sweet-expressions. For example, the typical structure:

```
(cond ((condition1) (result1)) ((condition2) (result2)) ...
      (t (otherwiseresult)))
```

can be expressed the same way in sweet-expressions. But if you use I-expression indentation, they can become staggeringly more obvious to the uninitiated:

```
cond
 group (condition1)
   result1
 group (condition2)
   result2
 group t
   otherwiseresult
```

Since the infix operations are "merged" if they are all the same, this works as it would in mathematics in a sweet-expression:

```
 if (0 <= x <= 10)
   (dostuff) ; x is between 0 and 10 inclusive.
   (dontstuff) ; x is not between 0 and 10 inclusive.
```

Discarded Variation: Initial "(" on block disabling more

In the definition of Sweet-expressions above, any "(" disables indentation processing, but name-prefix and infix process continue to work. This is *mostly* compatible with existing Lisp code, but not completely. Name-prefixing is unlikely to cause a problem in well-formatted code, but expressions that *look* like infix expressions can certainly occur, and that could be a problem. Basically, it'd be very nice to be *able* to disable infix processing in some reasonable way.

I thought of one variation, which I describe here.. but then explain its problems.

One solution is to modify sweet-expressions (as described above) so that, if the *first* non-whitespace character of a newly-read block is "(", more of sweet-expressions could be disabled. We could disable infix processing, at least. Of course, once you do that, why not just disable everything and process it as a traditional s-expression? The advantage of doing this is that you can call the "original" read routine, with all of its local extensions. The rules would then get a little more complex to explain, but you also get massive backwards compatibility. If the first non-whitespace character is a special character (mainly ";", """, and ","), process them and try again. Otherwise, after than initial character normal sweet-expression processing occurs... so an embedded "(" in expressions like load("name") or if (n < 3) would work normally.

There are disadvantages to this approach, too. One sad thing is that parentheses-surrounded infix calculations would no longer work at the topmost level, e.g., "(3 + 4)" would work *inside* a block, but not as the topmost block. You could enter "3 + 4", but that would require two "Enters" if we keep the rule that "multiple terms on the initial line mean wait for more lines". The user could call using name-prefixing any function that returns its first parameter, e.g., if "calc" is a function that returns its first parameter, then "calc(3 + 4)" could do the calculation. An alternative for the "usual case" would be to add one more rule: if there are 3 or more terms on the first line, and it matches the infix pattern, then return immediately after the first line is entered - so "3 + 4" now works on the command line. I like that idea; it seems unlikely to happen by accident, and it's even simpler than "(3 + 4)". You'd still need a "calc()" function if on the initial line you have an infix operation whose first parameter is also infix, e.g., "(5 * 6) + 3" won't work - you'd have to use "calc(5 * 6) + 3". Such special-casing is not as "clean", unfortunately. The advantages in backwards-compatibility and ease-of-use are significant, though, so I'm leaning towards that change.

So here's how the rules would look:

> 1. If the first non-whitespace character of a new block is "(", it's a traditional s-expression; it may be prefixed by arbitrary numbers of blank lines (which may include spaces or tabs), ";...." comments, quoting ('), quasiquoting (`), or comma-lifting (,). Any spaces or tabs after the expression will be consumed before returning, to increase compatibility with older systems (an s-expression, followed by a bare atom, will be processed like a traditional Lisp processor would process it).
> 2. Otherwise, if (after skipping content-free lines) the line begins *without any* space or tab, and the line is either one complete term like load("...") or one complete infix expression like 5 + 6 + 7, it's considered a one-line sweet-expression; it's returned (and run) immediately at the end of the line. This special rule makes interactive use much more pleasant. Without this special rule, we have to enter extra blank lines all the time for sweet-expressions, even when it's "obvious" that a new line isn't needed. Note that more lines will *always* be requested if there are any unclosed parens.
> 3. Otherwise, it's a multi-line sweet-expression. Basically, putting tabs/spaces at the beginning of a sweet-expression forces a "multiple lines" meaning. While there's an open paren, indenting is ignored. If there's not an open paren on the first line, the second line's tab/space indent sets the "smallest indent"; any the line with less indentation (including a blank line) ends the sweet-expression block.

> The blank-line rule is needed to make interactive processing pleasant. I think a space-and-tab *only* line should be considered the same as an immediate-return blank line; you *could* make lines with only spaces and tabs have a different meaning, but that's certain to cause mysterious problems (since they'd *look* the same). Lines which include only a ;-prefixed comment should be completely ignored, even if they have less indentation, so that they can be included without ending the block.
>
> Note: If the first line was indented, but the second line is not indented at all, then the previous line is processed and returned, with the second line unconsumed. This deals with weird formats like:
>
> ```
>    hello
> (more stuff....)
> ```
>
> Such formats were probably intended to be interpreted as traditional s-expressions (they make no sense as multi-line sweet-expressions), so we'll interpret the first line as a traditional format to increase backwards compatibility.

The rules are complex, though. What's disturbing is that "(3 + 4) * 5" works completely differently from "3 + (4 * 5)"; that's the sort of inconsistency that makes a notation a burden instead of a pleasure. One complication is that the "read" function needs to return when it's read an expression, and ideally doesn't store internal state of whitespace alrady read. That means that recording the indentation of the initial block should be avoided, if it's reasonable to do so... otherwise we have to record that information.

In short, striving for backwards-compatibility, ease-of-use at the command line, ease-of-use for programming, readability, and full list processing without special-casing a lot of syntax involves tradeoffs... my goal is to find the "best" trade.

Variations: Using different grouping symbols

There are times when you don't want infix processing (or want to control it specially). Also, it's clearly *possible* that reading an existing s-expression, but using infix-interpretation, could occasionally *silently* change the meaning of that expression. And that is obviously of concern to some.

In discussions with Alan Manuel Gloria, another idea popped up: really using multiple different grouping/list markers. The original discussion noted using [] for grouping/list initialization instead of (). Basically, [...] would start a new list and turn off indentation, but continue name-prefix and infix processing. In contrast, () would at least disable infix processing.

Although the original discussions used [...], I tried out various experiments and think that {...} is a better choice. The problem is that [...] is used in mathematics, and many programming languages including Python and Haskell, to mean a literal list. But this is exactly what we *don't* mean - instead, we want a "grouping" symbol that means that we specially interpret what's inside (in particular, that we interpret infix operators). In contrast, all languages with C-like syntaxes (C, C++, Java, C#) have trained folks to read {..} as possibly meaning a block, and in mathematics {...} can denote sets (again, special interpretation used for what's inside).

So, for the moment, let's presume that unprefixed (...) will indicate "no infix" (since this is much more backwards-compatible with s-expressions). There are actually two alternative ways to interpret (), if it's to be distinct from {} grouping. One is to say that () disables only infix, but you still have name-prefixing, and you can switch back to infix inside by using {}. Another is that () switches to pure s-expression processing.

There's a big advantage to saying that () switches to *pure* s-expression processing (disabling both indentation and infix processing, as well as indentation). If you define it that way, the sweet-expression reader can immediately call the original built-in (read) routine instead. That's important - while implementing sweet-expressions, I've found that there's a lot of implementation-unique read processing that you have to try to simulate if you want ALL the functionality of the underlying system. (E.G., standard Scheme doesn't have |..| constants, but some implementations do, so to read stuff for their Scheme you have to implement these new oddities.) That makes it essentially impossible to write a portable implementation of sweet-expressions that retains the capabilities of the underlying system. As a side-benefit, most s-expression files would be just readable as-is... which I admit is NOT a 'most important' criteria, but it'd be nice to have.

On the other hand, sometimes you'd like to say "no infix or indent processing", but be able to use name-prefixing, and switch to infix inside it. Guess what - there's a symbol pair already with that meaning: [..]. So [f g(x) + h(x)] is interpreted as (f (g x) + (h x)), without any infix operator in sight. Let's say that [..] has this meaning as long as it's *not* name-prefixed; that way, we can later give expressions like name[i] another meaning. If we use [...] to mean "list, no infix" and {...} to mean "list, with infix", you can then write sweet-expressions without (...) at all. This means that we could then give the reader a choice of how to handle (...)... as grouping, like {}, or as no infix, as []. The latter is more backwards-compatible, and so I'd expect that to be the default; the former is much more similar to other languages.

I think it'd be nicer to start with "infix as the default" (maybe a callable parameter if you want something else). That's a reasonable default simply because MOST people are more comfortable with it. Since ANY open paren disables the infix until its matching paren, users that use ordinary s-expressions might not even notice sweet-expressions. It's not like normal Lispers type in "3 + 4" and expect it to work

:-). So it won't harm those who want s-expression prefix, but it will help those who want infix. That might make sweet-expressions more likely to be accepted; you could enable them, and yet for most people they wouldn't even notice the difference (except for the *addition* of new abilities).

The rules are simple: When unprefixed, use {} for grouping, which disables only indentation; use [...] to disable indentation and infix; use () to disable everything (infix, name-prefix, and indentation). A prefixed {} or () is a function call, and only disables indentation. We'll reserve prefixed [...] for future use. This use of {} instead of () for grouping infix operators like {4 + {3 * 2}} is _slightly_ nonstandard, but not terribly so, and lets us be completely backwards-compatible with s-expressions. That's pretty compelling.

What about name-prefixed forms? Even if (f x) disables infix processing, that does not mean that f(x) must do the same. It's consistent in one sense if it does, but f{x} looks really funny while f(x) looks very conventional. I'm presuming, for the moment, that unprefixed (...) will really only be used for backwards-compatibility.

So now we're back to nice forms, without any calc()-like irregularities:

```
3 + 4
3 + {5 * 6}         ; just use {} instead of ().
{3 + 5} * 6         ; just use {} instead of (). No special cases.
{3 + 4} * f(x + y)  ; Function calls work fine.
(+ 3 4) * f{x + y}  ; this works too (!)
f{3 + 4}            ; Function calls support infix, is (f (+ 3 4))
f(3 + 4)            ; Proposal: prefixed () keeps infix, so it's (f (+ 3 4))
[1 + 2]             ; Here's a safe way to describe (1 + 2).
[f 3 + 4]           ; Here's (f 3 + 4).
(f 3 + 4)           ; Same thing.
```

There's another interesting benefit too. This would mean that sweet-expressions would be full of {...} when infix is used... which would make them even easier to tell apart from s-expressions.

Interestingly, this also makes it easy to use special "infix" macros (often written as "nfx"). Just tell users that "for advanced infix processing, use (nfx ....) and then follow these infix rules (which can then support precedence levels, defining infix operators, etc.)". So now you can have a choice: (1) a built-in "base" system doesn't need to be told about operators, precedence, or associativity, but you have to group all different operators using [] and you can only use punctuation as infix operators... and (2) advanced "nfx-like" functions, but one where you have to define all that stuff.

In theory, a program written using Scheme RSR6 might use [...] to mean the same as (..). But I bet almost no one actually DOES that.

This switch to using {...} everywhere for grouping, and (...) for straight s-expressions, is very tempting. The factorial example becomes:

```
defun factorial {n}          ; Parameters can be indented, but need not be
  if {n <= 1}                ; Supports infix, prefix, & function <=(n 1)
      1                      ; This has no parameters, so it's an atom.
      n * factorial(n - 1)   ; Function(...) notation supported
                             ; Function{...} okay too.
```

It's easy to accidentally use parens when you meant to use {...}, particularly in expressions like {4 + {3 * 2}}. But it's still tempting to make this change, since this makes the entire notation even more backwards-compatible, as well as giving it an easy (and clear) way to disable infix.

Since this gives an easy way to disable infix, this does raise the possibility of allowing "and" and "or" as built-in infix operators. After all, now infix operators are easy to disable. On the one hand, it's horribly inconsistent to have "and" and "or" built-in as infix, since all other infix operators are punctuation. On the other hand, "and" and "or" are essentially supported by all major Lisps. Also, these two operators are often treated specially; provers often treat them specially, and some